Carl von Ossietzky Universität Oldenburg
Department für Informatik
Abteilung Informationssysteme
Prof. Dr. Hans-Jürgen Appelrath

Individuelles Projekt

Implementierung und Visualisierung von Entwurfsmustern im Java-Hamster-Modell

vorgelegt von: Florian Marwede

Erstprüfer: Dr. Dietrich Boles

Zweitprüferin: Dr. Susanne Boll

Oldenburg, den 7. September 2005

Inhaltsverzeichnis

1 Einleitung

Entwurfsmuster sind „einfache und elegante Lösungen für spezifische Probleme des objektorientierten Softwareentwurfs.“[1] Sie basieren auf den Erfahrungen professioneller Software-Entwickler und ihre Güte ist somit in jahrelanger Praxis gezeigt, allerdings nicht in der Theorie bewiesen worden.
Durch gezielten Einsatz objektorientierter Techniken, wie etwa der Polymorphie, sollen Entwurfsmuster einerseits eine größere Wiederverwendbarkeit und Flexibilität von Software erreichen und andererseits durch einen prägnanten Namen das Entwurfsvokabular bereichern, die Kommunikation zwischen Entwicklern beschleunigen und ihr Verständnis füreinander erhöhen.
Anstatt, dass Entwickler A einige Diagramme für Entwickler B zeichnet und ihm dann umständlich seine Lösungsideen erklärt, kann er vorschlagen, man könne doch einen *Dekorierer* einsetzen. Mit der Voraussetzung, dass beide dieses Entwurfsmuster kennen, ist beiden nur mit der Nennung dieses abstrakten Begriffs klar, was genau gemeint ist.
Dennoch soll der Eindruck vermieden werden, Entwurfsmuster seien ein Allheilmittel gegen Probleme des Softwareentwurfs oder dass durch sie eine Entwurfsdiskussion sich immer auf die Nennung einzelner Muster reduzieren würde. Aus diesem Grunde ist neben der Kenntnis der Struktur der einzelnen Entwurfsmuster auch das Erkennen von Situationen, in denen sie eingesetzt werden können, von entscheidener Bedeutung. Diese Arbeit soll zu beidem einen Beitrag leisten.

1.1 Motivation und Ziele

Ziel dieses Individuellen Projektes ist die Demonstration von Entwurfsmustern mit dem Hamstermodell.
Dies soll auf zwei verschiedene Arten geschehen: Einerseits will ich versuchen, Entwurfsmuster durch Visualisierung mit geeigneten Hamsterprogrammen und dem Hamstersimulator begreifbar zu machen. Andererseits werde ich an einem etwas größeren praktischen Beispiel – die Entwicklung eines Frameworks für 2-Personen-Strategiespiele – zeigen, wie man mehrere Entwurfsmuster gemeinsam im objektorientierten Softwareentwurf einsetzen kann.

Das Hamstermodell wurde von Dr. Dietrich Boles entwickelt, um spielerisch möglichst viele Bereiche der Software-Entwicklung zu erklären – von der imperativen über die objektorientierte Programmierung bis zu Elementen des objektorientierten Softwareentwurfs. In den letzten Bereich ist meine Arbeit einzuordnen.
Meiner persönlichen Erfahrung nach gibt es hauptsächlich Lehrwerke, die sich entweder mit den Grundlagen der Programmiersprachen an Anfänger richten, oder in groben, verallgemeinernden Zügen die Theorie des System- und Architekturentwurfs beschreiben. Für jemanden, der gerade einen Programmierkurs hinter sich hat, bieten die einen zu wenig und die anderen zuviel Information an – Entwurfsmuster gehören zu den Themen, welche die Lücken dazwischen füllen.
Meine Hauptzielgruppe sind damit Schüler und Studenten, die zwar mit den Grundlagen der objektorientierten Programmierung vertraut sind, aber wenig oder gar keine Praxiserfahrung in der Softwareentwicklung haben. Ihnen möchte ich mit den Entwurfsmustern Werkzeuge dazu in die Hand geben und diese am konkreten Beispiel – der Entwicklung des Spiele-

[1] siehe [GHJV04]

Frameworks – demonstrieren. Weiterhin werde ich hier Möglichkeiten für den Leser offen lassen, neben den zwei Spielen, die ich selber implementieren werde, weitere hinzuzufügen.
Weitere Zielgruppe sind Programmierer, die sich zwar schon auf fortgeschrittenerem Niveau befinden, doch noch nicht – oder nicht bewusst – mit Entwurfsmustern gearbeitet haben. Das Hamster-Modell ist leicht verständlich, und damit auch für Quereinstieger geeignet.

Neben der Erläuterung der Entwurfsmuster ist es mein Ziel, das Verständnis für herkömmliche, häufig eingesetzte Programmiertechniken zu vertiefen. Zum Beispiel werden zwar die Grundlagen der Ausnahmebehandlung in einem Programmierkurs behandelt, doch ist damit nicht zugleich das Wissen geschaffen, wann genau man Exceptions einsetzt und wie man sich aus der Klassenbibliothek bedient. Auch hat ein Anfänger der Programmiersprache Java im Prinzip das Wissen darum, dass in einem Interface Konstanten definiert werden dürfen. Aber es ist unwahrscheinlich, dass er daraus schließen wird, wie sinnvoll es in großen Klassenstrukturen sein kann, Konstanten bestimmter Bereiche in nur für diesen Zweck geschaffenen Interfaces zu deklarieren, die dann von den Klassen implementiert werden können.
Genau wie Entwurfsmuster sind auch dies alles Dinge, auf die man auch von alleine kommen kann, wenn man die Grundlagen einer objektorientierten Programmiersprache beherrscht. Eine Arbeit wie diese soll nur helfen, Lernprozesse zu beschleunigen und Entwurfsmaterial zur Reflexion bereitstellen.

1.2 Anmerkungen für den Leser

1.2.1 Sprache und Namenskonventionen

Ich werde versuchen, so weit es möglich ist, im Text in der deutschen Sprache zu verbleiben, werde also insbesondere deutsche Namen für die Entwurfsmuster verwenden. Trotzdem verwende ich übliche Namenskonventionen, wenn das Englische allgemein gebräuchlich ist und deutsche Übersetzungen nur verwirrend wären, z.B. bei set/get-Methoden oder Exceptions.

1.2.2 Formales

- Wenn ein Begriff zum ersten Mal verwendet bzw. eingeführt wird, so ist er *kursiv* gedruckt.

- Wenn ein Wort im Fließtext in derselben Schriftart gedruckt ist, wie der Quellcode, dann ist damit auch eine Klasse oder eine Methode aus dem Quelltext gemeint. Ein Beispiel ist `ZaehlHamster`. Im Gegensatz dazu ist übrigens mit dem Wort „Zählhamster“ der virtuelle Hamster selbst bzw. die Grafik, die ihn repräsentiert, gemeint.

- Für den Fall, dass eine vorgestellte Klasse variiert erneut dargestellt wird, werde ich bei den Methoden, die sich nicht geändert haben, den Methodenrumpf aussparen und durch {...} ersetzen. Diese Vorgehensweise wird ebenfalls angewandt, wenn sich zwar nicht ganze Methoden, aber große Programmteile nicht geändert wiederholen.

1.2.3 Verwendung von Java-Version 5.0

Ich verwende Java 5, an Neuerungen gegenüber Java 1.4 aber nur die vereinfachte Form der `for`-Schleife und die simpelste Form der Generics. Dies sollte der Leser also kennen.

1.3 Aufbau der Arbeit

In Kapitel 2 werden die Konzepte und Werkzeuge beschrieben, die dieser Arbeit zu Grunde liegen:

- Das Hamstermodell
- Entwurfmuster
- UML
- Die Java-Klassenbibliothek

Je nach Kenntnisstand des Lesers können diese ausgelassen werden – jemand, der gerade [Bol02] und [BB04] durchgearbeitet hat, wird in Abschnitt 2.1 nichts Neues finden.
In Kapitel 3 werden die einzelnen Entwurfsmuster – in steigendem Schwierigkeitsgrad angeordnet – erläutert. Dabei werden eigens dafür konzipierte Hamsterprogramme verwendet, die dem Leser auch zum Experimentieren und Weiterentwickeln in Dateien zur Verfügung stehen.
In Kapitel 4 wird die Entwicklung eines Spiele-Frameworks gezeigt. Dabei werden die Themen Anforderungsdefinition, Entwurf und Prototypenentwicklung behandelt. Aus grafische Oberfläche (View) wird der Hamstersimulator verwendet.
In Kapitel 5 wird beispielhaft gezeigt, wie man das Framework nutzen kann, um das Damespiel zu implementieren.
Auf der beiliegenden CD ist sämtlicher Quellcode, der in dieser Arbeit erläutert wurde, enthalten.

2 Grundlagen

2.1 Das Hamster-Modell

Das von Dr. Dietrich Boles entwickelte Hamster-Modell besteht aus drei wesentlichen Komponenten:

1. In einer Lehrbuchreihe wird das Programmieren mit der Sprache Java vermittelt. Dabei werden die einzelnen Konzepte von Java getrennt vorgestellt und damit die Komplexität der Sprache für den Lernenden reduziert.

   Die Besonderheit dieser Werke ist, dass die Programmiersprache nicht nur gezeigt und erläutert wird. Mit vielen Beispielen und Übungsaufgaben in jedem Kapitel wird es dem Leser möglich gemacht, auf spielerische Art und Weise die gezeigten Konzepte *selbst* auszuprobieren und das Programmieren einzuüben. Um diese Beispiele und Aufgaben so anschaulich und damit so effektiv wie möglich zu machen, wird hierbei das Hamster-Modell eingesetzt: Ein virtueller Hamster läuft durch ein virtuelles Territorium und muss dabei Aufgaben lösen. Durch diese „Hamsteraufgaben“ werden dem Leser nicht nur programmiersprachenspezifische Konzepte vermittelt, sondern auch das für das Programmieren unabkömmliche logische, problemorientierte Denken.

   Bisher sind in dieser Reihe [Bol02] und [BB04] erschienen, welche die Grundlagen der imperativen und objektorientierten Programmierung zum Ziel haben. In Band 3 – in dem es um parallele Programmierung gehen soll – wird voraussichtlich Ende des Jahres 2006 erscheinen.

2. Der von Daniel Jaspers entwickelte Hamster-Simulator ist ein Programm, mit dem Hamster-Programme erstellt, ausgeführt und getestet werden können. Somit ist es immer direkt möglich, das in einem Kapitel Gelernte sofort in die Praxis umzusetzen und zu trainieren. Diese erste Entwicklungsumgebung stellt dem Programmieranfänger einen Editor mit Syntay-Highlighting, einen Verzeichnis-Browser, einen Compiler und einen Debugger, sowie eine grafische Oberfläche zur Verfügung, in der die Hamsterprogramme bei Ausführung sichtbar gemacht werden können.

3. Die Webseite des Hamster-Models[2] informiert über neue Veröffentlichungen, stellt den Hamster-Simulator zum Download bereit und unterhält ein Forum, in dem Leser fragen stellen und Lösungen zu Hamsteraufgaben diskutieren können.

Das Hamster-Modell ist seit vielen Jahren im Programmierkurs Java der Carl von Ossietzky-Universität Oldenburg erfolgreich im Einsatz und hat damit gezeigt, dass es praxistauglich und benutzerfreundlich ist.

Alle Hamsterprogramme basieren auf der Oberklasse `Hamster`, da alle anderen Hamster von ihr erben. Sie ist zusammen mit der Klasse `Territorium` und einigen „hamsterspezifischen“ Exceptions im Hamstersimulator verankert. All diese Klassen sind im Anhang von [BB04] beschrieben.
Eine Instanz der Klasse `Hamster` versteht die Grundbefehle `vor`, `linksUm`, `nimm`, `gib`, `kornDa`, `vornFrei` und `maulLeer`.

[2] www.java-hamster-model.de

2.1.1 Ein AllroundHamster

Um weitere häufig verwendete Methoden nutzen zu können, übernehme ich für diese Arbeit die Idee einer erweiterten Klasse `AllroundHamster` , von der alle in dieser Arbeit verwendeten Hamster erben (siehe [BB04]).
Bevor als nächstes der Quellcode des `AllroundHamster` folgt, ist zu den Parametern des Konstruktors zu bemerken, dass diese abgekürzte Schreibweise in [BB04] durchgängig verwendet und damit in dieser Arbeit übernommen wird. Die Abkürzungen stehen für:

- r : Reihe
- s : Spalte
- b : Blickrichtung
- k : Körneranzahl

```
package entwurfsmuster;

public class AllroundHamster extends Hamster {

    public AllroundHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }

    public void vorUndNimm () {
        if (vornFrei()) {
            vor();
            if (kornDa()) {
                nimm();
            }
        }
    }

    public void kehrt() {
        linksUm();
        linksUm();
    }

    public void rechtsUm() {
        kehrt();
        linksUm();
    }

    public boolean linksFrei() {
        linksUm();
        boolean frei = vornFrei();
        rechtsUm();
        return frei;
    }
```

```
    public boolean rechtsFrei() {
        rechtsUm();
        boolean frei = vornFrei();
        linksUm();
        return frei;
    }

    public void setzeBlickrichtung(int richtung) {
        while (getBlickrichtung() != richtung) {
            linksUm();
        }
    }

    public void gotoKachel(int reihe, int spalte) {
        if (reihe > getReihe()) {
            setzeBlickrichtung(Hamster.SUED);
        } else {
            setzeBlickrichtung(Hamster.NORD);
        }
        while (reihe != getReihe()) {
            vor();
        }
        if (spalte > getSpalte()) {
            setzeBlickrichtung(Hamster.OST);
        } else {
            setzeBlickrichtung(Hamster.WEST);
        }
        while (spalte != getSpalte()) {
            vor();
        }
    }
}
```

2.2 Entwurfsmuster

Christopher Alexander hat wohl als erster Entwurfsmuster definiert und beschrieben, allerdings nicht für Software, sondern für Städte und Gebäude (siehe [AIS+77]). Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides – in der Fachwelt auch als „Gang of Four“ (GoF) bekannt – griffen Alexanders Idee für die Software-Welt auf, sammelten und ordneten das hier vorhandene Material und gaben 1995 ihr Buch „Design Patterns“ heraus, das die ersten 23 „offiziellen“ Entwurfsmuster für den Softwareentwurf enthielt. Seitdem hat sich auf dem Gebiet viel getan und gerade im Internet sind viele neue Entwurfsmuster beschrieben und auch kritisch diskutiert worden[3] . Das Werk von Gamma, Helm, Johnson und Vlissides hat jedoch nie den Ruf als *das* Standardwerk auf diesem Gebiet verloren, deshalb ist die neueste Auflage der deutschen Übersetzung der „Design Patterns“ Grundlage meiner Arbeit (siehe [GHJV04]).

[3] siehe z.B. http://hillside.net/patterns/ oder http://www.oopsla.org

Im Folgenden möchte ich durch eine Reihe von Fragestellungen den Begriff des Entwurfsmusters dem Leser ein wenig näher bringen.

2.2.1 Wie sind Entwurfsmuster im Gesamtprozess der Software-Entwicklung einzuordnen?

Entwurfsmuster sind vor allem für den Feinentwurf gedacht, es gibt aber auch Abweichungen. Denn das Singleton-Muster ist schon fast auf die Ebene der Implementierung einzuordnen, da es nur eine Klasse betrifft und auch gut nachträglich zu verwenden ist. Das Fassaden-Muster hingegen soll ganze Subsysteme kapseln, ist also eher auf gröberer Ebene anzusiedeln.

2.2.2 Was für andere Arten von Mustern gibt es?

Muster, welche die Organisation ganzer Softwaresysteme beschreiben, nennt man *Architekturmuster*. Das Architekturmuster, welches ein Softwaresystem dominiert, wird auch als *Architekturstil* bezeichnet. Wenn ein Muster hingegen auf eine bestimmte Programmiersprache beschränkt ist, so spricht man von einem *Idiom*.
Eine Einführung in Architekturmuster, Entwurfsmuster und Idiome und deren Unterschiede und Zusammenhänge wird in [BMRSM00] gegeben. Mehr zu Softwarearchitektur und Architekturstilen findet man in [PBG04].

2.2.3 Wie lassen sich Entwurfsmuster beschreiben?

Zu der Beschreibung eines Entwurfsmusters gehört nicht nur die Struktur und ein prägnanter Name, sondern auch eine Beschreibung bekannter Vor- und Nachteile, sowie eine Referenzierung bekannter Verwendungen, um den erfolgreichen Einsatz des Musters in der Praxis zu belegen.
Verschiedene Ansätze zur Beschreibung und Katalogisierung von Entwurfsmustern sind bei [GHJV04], [BMRSM00] und [BMMM98] zu finden.

2.2.4 Wie können Entwurfsmuster klassifiziert werden?

In [GHJV04] werden die Entwurfsmuster zum einen nach Kategorien „klassenbasiert“ und „objektbasiert“ und zum anderen nach den Kategorien „Erzeugungsmuster“, „Strukturmuster“ und „Verhaltensmuster“ eingeteilt.

Dieser Abschnitt soll dabei helfen, eine deutlichere Idee davon zu gewinnen, was ein Entwurfsmuster eigentlich ist – aber auch zeigen, dass keine klare, hundertprozentig eindeutige Definition möglich ist, die entscheiden lässt, was ein Entwurfsmuster ist und was nicht. Sehr wahrscheinlich benutzen viele erfahrene Entwickler bestimmte Entwurfsmuster, ohne es zu wissen bzw. ohne den in der Fachwelt etablierten Namen zu kennen.
Im folgenden Abschnitt wird beispielhaft erläutert, welche Denkstruktur sich hinter Entwurfsmustern verbirgt.

2.2.5 Sinn und Zweck von Entwurfsmustern

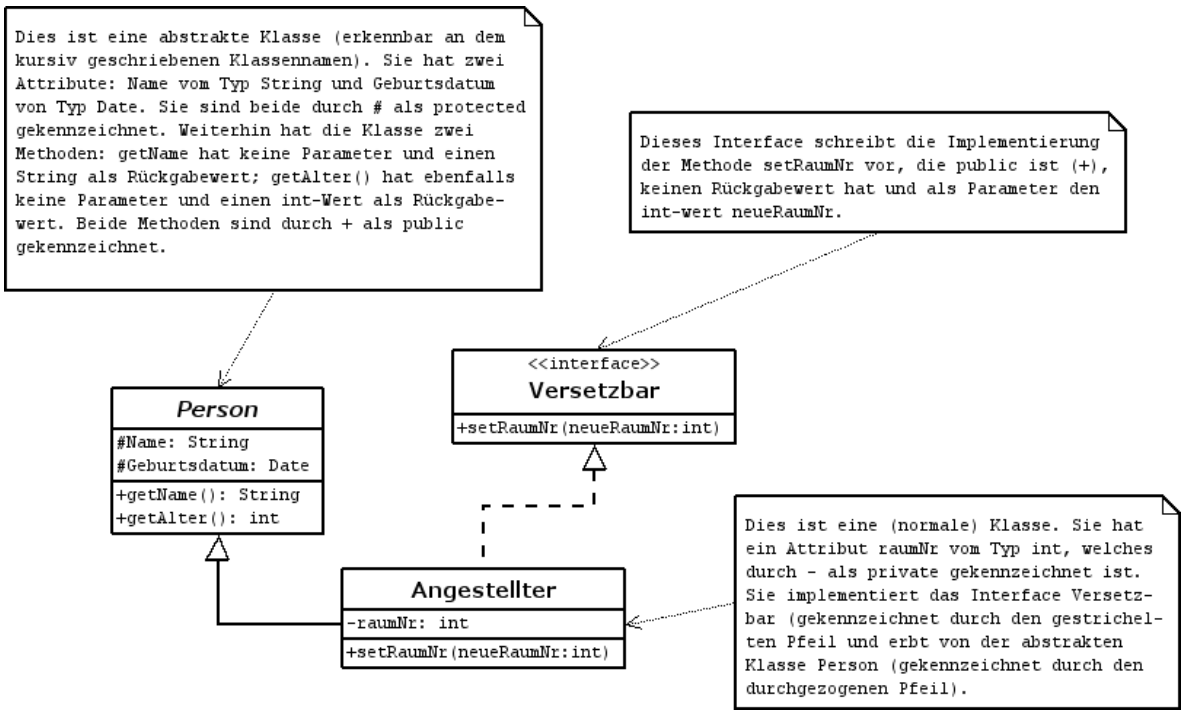
Manchmal wird in der Mathematik ein bestimmter Aufgabentyp vorgerechnet, man begreift das Prinzip und kann andere Aufgaben nach demselben Prinzip lösen, ohne viel darüber nachzudenken.

Mit zunehmender Komplexität können Matheaufgaben aber nicht mehr nach einem Schema „maschinell“ gelöst werden. Natürlich sind sie immer noch lösbar, allein durch die Anwendung von Rechengesetzen usw., aber sie erfordern ein gewisses Maß an Kreativität insofern, als dass man jene Rechengesetze und Vorgehensweisen geschickt kombinieren muss, um zum Ziel zu gelangen. Aus meiner persönlichen Erfahrung heraus kann ich sagen, dass gerade diese Fähigkeit die Menschen, denen Mathe Spaß macht von denen trennt, denen das nicht so geht.
Bei der Softwareentwicklung ist es nun ähnlich: Es gibt kein Schema F, das komplexe Entwurfsentscheidungen automatisch richtig fällt, aber es gibt bestimmte Vorgehensweisen, die sich erfahrene Programmierer mit der Zeit aneignen (ebenso wie die Fähigkeit, diese geschickt zu kombinieren), um zumindest eine elegante Lösung von vielen zu finden.
Es wird meistens so sein, dass es mehrere Lösungen gibt, gerade, wenn das Problem komplex ist, und es wird nicht immer eindeutig entscheidbar sein, welche von zwei Lösungen, die bessere ist. Aber man kann schon grob von besseren und schlechteren Lösungen sprechen und Ziel dieser Arbeit ist es zu zeigen, wie man auf die besseren Lösungen kommen kann.

2.3 UML - Unified Modelling Language

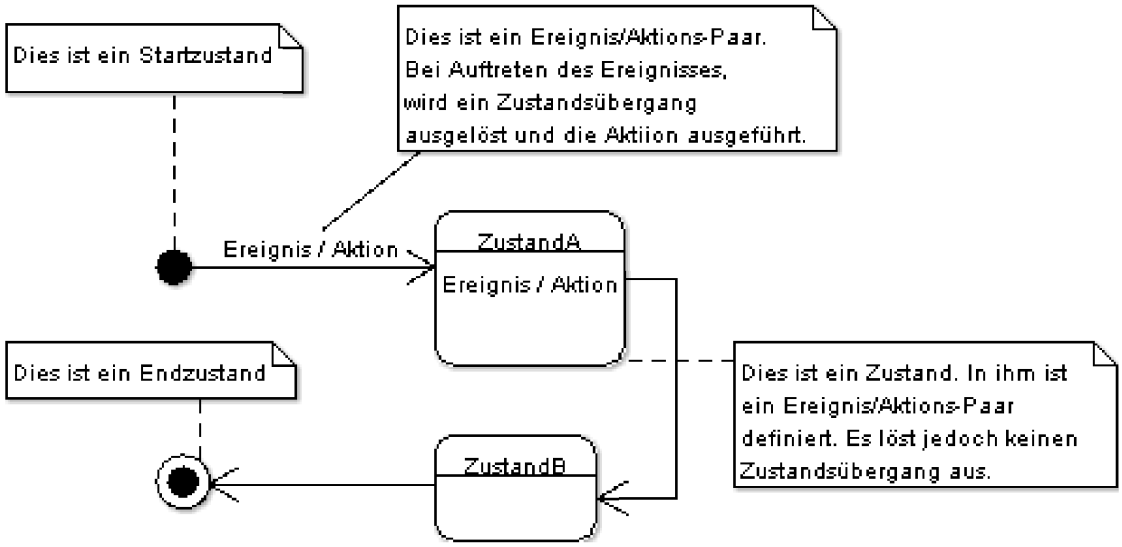
Die vollständige Spezifikation der UML 2.0 wird von der OMG im Internet zum Download bereitgestellt.[4] Eine Einführung bietet [ZGK04].
In dieser Arbeit werden nur das Klassendiagramm und das Zustandsdiagramm der UML verwendet, deren Anwendung im Folgenden durch zwei selbsterklärende Diagramme gezeigt wird:

2.3.1 Klassendiagramm



[4] www.omg.org/cgi-bin/doc?ptc/2004-10-02

2.3.2 Zustandsdiagramm



2.4 Einführung der Java-Klassenbibliothek

Ein Vorteil der Programmiersprache Java ist, dass viel Mühe auf eine gute Programmierbibliothek verwendet wurde, die alle wichtigen Funktionen in den zentralen Bereichen Datenstrukturen, Ein- und Ausgabe, Grafik- und Netzwerkprogrammierung abdeckt.
Im Folgenden liste ich die Klassen auf, die ich aus der Java-Klassenblibliothek verwende: ArrayList, HashMap und Math. Ich werde jeweils zunächst kurz beschreiben, was die Klasse kann und wofür sie eingesetzt wird, danach werde ich in verkürzter Form die Klasse mit ihren Methoden vorstellen (allerdings nur die Methodenköpfe, die Implementierung soll uns nicht interessieren).

2.4.1 ArrayList

Eine `ArrayList` ist eine Mischung aus einem Array [5] und einer Liste. Von einer Liste sprechen wir in Java, wenn eine Klasse das Interface `List` implementiert.
In diesem Zusammenhang ist es sinnvoll, einen Blick auf das „Collection-Framework“ zu werfen, welches wohl eines der bekanntesten und am häufigsten eingesetzten Teile der Java-Klassenbibliothek ist. In Abbildung 1 ist ein Ausschnitt aus der Klassenhierarchie zu sehen.
Das Interface `Collection` steht für eine Gruppe von Objekten und bietet grundlegenste Operationen auf ihnen an, wie z.B. „Objekt hinzufügen“ mit der Methode `add` oder „Objekt entfernen“ mit der Methode `remove`. Die Klasse `AbstractCollection` ist hierfür eine Basisimplementierung.
`List` wird nun von `Collection` abgeleitet und steht für eine geordnete Gruppe von Objekten, d.h. es ist zum Beispiel ein direkter Zugriff auf ein Objekt über einen Index möglich.

[5] Arrays werden ausführlich beschrieben in [BB04] S.199ff (Kapitel8)

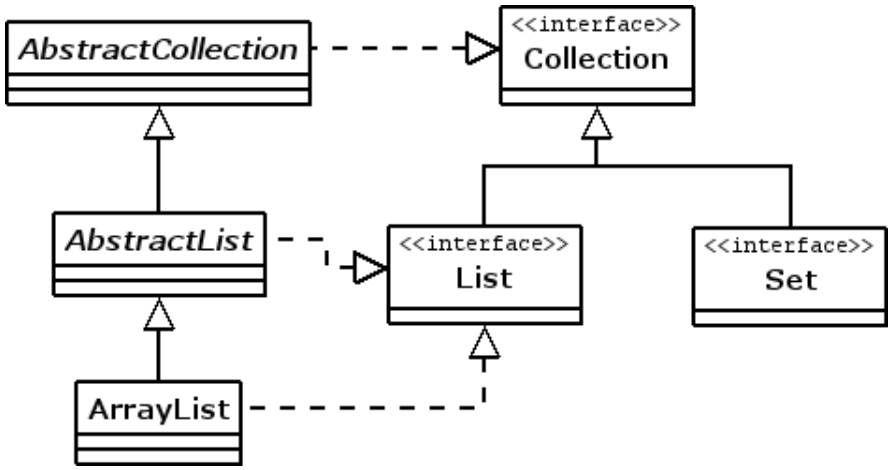


Abbildung 1: Ausschnitt aus dem Collection-Framework

Eine andere Spezialisierung der `Collection` bietet das `Set`, welches die mathematische Menge repräsentieren soll; auffälligstes Merkmal ist hier, dass kein Element doppelt vorkommen darf – oder präziser gesprochen: Wenn man versucht, zu einem `Set` eine Referenz auf ein bestimmtes Objekt hinzuzufügen und eine Referenz auf dieses Objekt schon in dem `Set` vorhanden ist, wird das `Set` unverändert gelassen (im Gegensatz dazu sind bei einer `List` doppelte Referenzen erlaubt).
Parallel zu der `AbstractCollection` gibt es die `AbstractList` als Basisimplementierung von `List` – und `ArrayList` ist schließlich eine Unterklasse von `AbstractList`, die zur internen Speicherung der Elemente ein Array benutzt.

Soweit zur Theorie – um die Programmierbeispiele in dieser Arbeit zu verstehen, muss man aber nicht das gesamte Collection-Framework durcharbeiten (von dem das bis jetzt Beschriebene nur eine sehr kleine Auswahl ist); es genügt, sich zu merken, dass die `ArrayList` in der Hauptsache ein Array mit veränderbarer Größe ist (weitere Eigenschaften sollen uns hier nicht interessieren). Auf eine `ArrayList` kann man aber nicht – wie bei einem Array – mit der Kurzschreibweise der eckigen Klammern zugreifen, sondern es werden andere Methoden zur Verfügung gestellt, mit denen man dies erreichen kann. Die Methodenköpfe der wichtigsten davon sind in der folgenden (verkürzten) Darstellung der Klasse beschrieben.

```
package java.util;
/**
 * Bemerkung zum Klassenkopf: E wird bei einer konkreten Implementierung
 * durch die Klasse ersetzt von deren Typ die Elemente der ArrayList
 * sein sollen.
 * Die implementierten Interfaces werden weiter unten erklärt.
 */
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    /**
     * Erzeugt eine leere ArrayList mit der übergebenen Kapazität.
     */
```

```
public ArrayList(int initialCapacity);
/**
 * Erzeugt eine leere ArrayList mit einer initialen Kapazität von
 * zehn Elementen.
 */
public ArrayList();
/**
 * Gibt die Anzahl der in der ArrayList enthaltenen Elemente zurück.
 */
public int size();
/**
 * Testet, ob die ArrayList leer ist.
 */
public boolean isEmpty();
/**
 * Testet, ob ein bestimmtes Element in der ArrayList ist.
 */
public boolean contains(Object elem);
/**
 * Sucht den Index eines übergebenen Objektes.
 */
public int indexOf(Object elem);
/**
 * Gibt eine Kopie der ArrayList zurück.
 */
public Object clone();
/**
 * Gibt ein Array zurück, dass alle Elemente der ArrayList enthält.
 * Der Typ des Arrays ist der von dem als Parameter angegebenen Array.
 * Wenn die Elemente der Liste in dieses übergebene Array passen,
 * wird dies zur Rückgabe benutzt. Andernfalls wird ein neues erzeugt.
 */
public <T> T[] toArray(T[] a);
/**
 * Gibt das Objekt an der übergebenen Position zurück.
 */
public E get(int index);
/**
 * Ersetzt das Element an der angegebenen Position durch das angegebene
 * Element.
 */
public E set(int index, E element);
/**
 * Fügt das übergebene Objekt der ArrayList an.
 */
public boolean add(E o);
/**
 * Entfernt das Objekt an der angegebenen Position aus der ArrayList.
```

```
     */
    public boolean remove(Object o);
    /**
     * Entfernt alle Objekte aus der ArrayList.
     */
    public void clear();
    /**
     * Fügt alle Elemente der angegeben Collection an die ArrayList an.
     * Diese Elemente müssen vom Typ der Klasse der Elemente der ArrayList
     * sein oder von ihr erben.
     */
    public boolean addAll(Collection<? extends E> c);

}
```

Die drei Interfaces `RandomAccess`, `Clonable` und `Serializable` sind sogenannte *Marker-Interfaces*[6] damit leer implementiert – sie dienen nur dazu, um anzuzeigen, dass die implementierende Klasse ein bestimmtes Verhalten erfüllt.[7]

2.4.2 Hash Map

Eine `Map` ist ein Objekt, das „Schlüsselobjekte“ mit „Wertobjekten“ verbindet, d.h. es speichert Paare mit jeweils einem von diesen Elementen und man kann z.B. unter Angabe des Schlüsselobjektes auf das Wertobjekt zugreifen. Anschaulich ist hier der Vergleich mit einem Telefonbuch. Wenn man den Schlüssel – hier: den Namen einer Person – kennt, kann man darüber den entsprechenden Wert – hier: die Telefonnummer der Person – erfragen.
Das Interface `Map` und die zugehörigen Klassen sind ebenfalls Teil des Collection-Framework. In dieser Arbeit wird nur die auf einer Hashtabelle[8] basierende `HashMap` verwendet, die von `AbstractMap` erbt. `AbstractMap` ist parallel zur `AbstractList` eine Basisklasse, die viele Methoden defaultmäßig implementiert, um das Verwenden des Interface `Map` zu erleichtern.

```
package java.util;
import  java.io.*;
/**
 * Bemerkung zum Klassenkopf: K wird bei einer konkreten Implementierung
 * durch die Klasse ersetzt, von deren Typ die Schlüssel der ArrayList
 * sein sollen, V für jene, die den Werten zugeordnet sind.
 */
public class HashMap<K,V> extends AbstractMap<K,V>
                          implements Map<K,V>, Cloneable, Serializable {
    /**
     * Die Anzahl der Schlüssel-Wert-Paare, die die HashMap enthält.
     */
    transient int size;
    /**
     * Erzeugt eine neue Instanz einer HashMap.
```

[6] [Ess04] Kapitel 6.1
[7] ausführliche Beschreibung siehe http://java.sun.com/j2se/1.5.0/docs/api/index.html
[8] siehe z.B. http://de.wikipedia.org/wiki/Hash-Tabelle

```
 */
public HashMap();
/**
 * Erzeugt eine HashMap mit denselben Schüssel-Wert-Paaren
 * der angegeben Map.
 */
public HashMap(Map<? extends K, ? extends V> m);
/**
 * Gibt die Anzahl an Schlüssel-Wert-Paaren in dieser HashMap zurück.
 */
public int size();
/**
 * Gibt true zurück, wenn die HashMap leer ist, sonst false.
 */
public boolean isEmpty();
/**
 * Gibt das Wert-Objekt zurück, das zum angegeben Schlüssel gehört.
 * Wenn es kein Wert-Objekt zum angebenen Schlüssel gibt, wird null
 * zurück gegeben. Dies geschieht allerdings auch, wenn zu dem
 * betreffenden Schlüssel als Wert explizit null angegeben ist.
 * Um diese beiden Fälle zu unterscheiden, kann die containsKey-Methode
 * benutzt werden.
 */
public V get(Object key);
/**
 * Gibt true zurück, falls ein Wert-Objekt für den angebenen Schlüssel
 * vorhanden ist, sonst false.
 */
public boolean containsKey(Object key);
/**
 * Fügt das angegebene Schlüssel-Wert-Paar in die HashMap ein.
 * Wenn es ein solches schon für den angegebenen Schlüssel gibt,
 * wird das alte durch das neue ersetzt.
 */
public V put(K key, V value);
/**
 * Kopiert alle Schlüssel-Wert-Paare der angegebenen Map in die HashMap.
 * Wieder werden bei gleichen Schlüsseln alte Schlüssel-Wert-Paare
 * durch neue ersetzt.
 */
public void putAll(Map<? extends K, ? extends V> m);
/**
 * Entfernt das Schlüssel-Wert-Paar zu dem angegeben Schlüssel
 * aus der HashMap.
 */
public V remove(Object key);
/**
 * Entfernt alle Schlüssel-Wert-Paare aus der HashMap.
```

```
     */
    public void clear();
    /**
     * Gibt true zurück, wenn es mindestens einen Schlüssel
     * in der HashMap gibt, der mit dem angebenen Wert-Objekt
     * assoziert ist, sonst false.
     */
    public boolean containsValue(Object value);
    /**
     * Gibt eine Kopie der HashMap zurück.
     */
    public Object clone();

}
```

2.4.3 Math

Die Klasse `Math` stellt einige häufig eingesetzte mathematische Methoden und Konstanten bereit, wie zum Beispiel die Winkelfunktionen und den natürlichen Logarithmus.
Auch ist hier mit der Methode `random` eine Möglichkeit gegeben, Zufallszahlen zu generieren, von der in dieser Arbeit mehrmals Gebrauch gemacht wird.

```
public final class Math {
    /**
      * E, die Basis für natürliche Logarithmen als Konstante.
      */
    public static final double E = 2.7182818284590452354;
    /**
      * PI, die Kreiszahl als Konstante.
      */
    public static final double PI = 3.14159265358979323846;
    /**
      * Die Winkelfunktionen.
      */
    public static double sin(double a);
    public static double cos(double a);
    public static double tan(double a);
    public static double asin(double a);
    public static double acos(double a);
    public static double atan(double a);
    /**
      * Zwei Methoden, um Radiantmaß in Gradmaß und umgekehrt
      * zu konvertieren.
      */
    public static double toRadians(double angdeg);
    public static double toDegrees(double angrad);
    /**
      * Gibt das Ergebnis der Potenz zurück, die die Eulersche Zahl
      * zur Basis und den angegeben Parameter a als Exponent hat.
```

```
      */
    public static double exp(double a);
    /**
      * Gibt den natürlichen Logarithmus des angebenen Parameters a zurück.
      */
    public static double log(double a);
    /**
      * Gibt die Quadratwurzel des angegebenen Parameters a zurück.
      */
    public static double sqrt(double a);
    /**
      * Gibt die Potenz zurück, die a zur Basis und b zum Exponenten hat.
      */
    public static double pow(double a, double b);
    /**
      * Zwei Methoden, um Fließkommazahlen auf Ganzzahlen zu runden.
      */
    public static int round(float a);
    public static long round(double a);
    /**
      * Gibt eine zufällige Zahl aus dem Intervall [0, 1) zurück.
      */
    public static double random();
    /**
      * Methoden für die Betragsfunktion.
      */
    public static int abs(int a);
    public static long abs(long a);
    public static float abs(float a);
    public static double abs(double a);
    /**
      * Methoden, die den größeren zweier Werte ermitteln.
      */
    public static int max(int a, int b);
    public static long max(long a, long b);
    public static float max(float a, float b);
    public static double max(double a, double b);
    /**
      * Methoden, die den kleineren zweier Werte ermitteln.
      */
    public static int min(int a, int b);
    public static long min(long a, long b);
    public static float min(float a, float b);
    public static double min(double a, double b);
}
```

3 Vorstellung von Entwurfsmustern an konkreten Beispielen

In diesem Kapitel werden in je einem Abschnitt ein Entwurfsmuster eingehend erläutert. Diese Abschnitte haben folgende Elemente gemeinsam:

- Es wird in einer oder mehreren Stufen gezeigt, wie man eine zu Beginn des Abschnitts gestellte **Motivationsaufgabe** mit dem jeweiligen Muster lösen kann. Sämtlicher gezeigter Quellcode ist auf der beigefügten CD enthalten.
- In den **Angaben zum Territorium** werden besondere Beschaffenheiten, die das Territorium aufweisen muss erläutert. Ein funktionierendes Beispielterritorium ist jeweils in dem Oberpacket eines Musters enthalten. Wenn die Rolle des `StandardHamster` nicht erwähnt wird, wird er auf seiner Startposition belassen und ignoriert.
- Nach der Lösung der Hamsteraufgabe folgen **Zusammenfassung und Ausblick**. Hier kann man – außer bei Singleton – ein Klassendiagramm finden, sowie die Klassifizierung nach [GHJV04] und einige praktische Hinweise für die Anwendung dies Musters.

Die einzelnen Abschnitte haben als Überschrift die Musternamen, die der Übersetzer von [GHJV04] gewählt hat. Wenn der englische Originalname davon abweicht, ist er in Klammern dahinter angegeben.
Die Pakete sind nach den Abschnitten geordnet, d.h.

- alle Programme zur Demonstration von Entwurfsmustern befinden sich in dem Paket `entwurfsmuster` (im Gegensatz dazu ist das Oberpaket für das Framework `spieleFramework`),
- alle Programme für das Beobachtermuster befinden sich in `entwurfsmuster.beobachter`,
- und alle Programme für die Lösung A für das Beobachtermuster befinden sich in `entwurfsmuster.beobachter.beobachterA`.
- Der oft eingesetzte `AllroundHamster` befindet sich in `entwurfsmuster`.

Beinahe alle Pakete enthalten eine ausführbare `main`-Methode, die die dem Abschnitt zugehörigen Klassen verwendet. Wenn eine Klasse sich bei einer Lösung nicht zu der Version der Vorgängerlösung verändert hat, wird sie von dort importiert.

3.1 Beobachter (Observer)

3.1.1 Motivationsaufgabe: Sammeln und Zählen

Ein Sammelhamster soll durch das Territorium laufen und Körner einsammeln. Wie geschickt er das tut – oder mit anderen Worten: wie gut der Algorithmus ist, der dahinter steht –, soll hier nicht weiter interessieren. Ein Zählhamster soll genau immer dann, wenn der Sammelhamster ein Korn gefunden hat, einen Schritt vorgehen. Der entscheidende Punkt an dieser Aufgabe ist die Realisierung der Kommunikation zwischen den Hamstern, d.h. woher der Zählhamster weiß, dass der Sammelhamster ein Korn eingesammelt hat.

3.1.2 Territorium

Am Besten ist ein Territorium ohne Mauern von kleiner bis mittlerer Größe, da die verwendete Suchstrategie zum Sammeln der Körner sehr zeitintensiv ist und man bei großen Territorien beim Testen sehr lange warten muss, um herauszufinden, ob das Programm korrekt abläuft. Natürlich sollten sich auch Körner im Territorium befinden.
Die Startposition des Sammelhamsters ist beliebig und der Zählhamster sollte so positioniert werden, dass er nach vorne hin „freie Bahn“ hat, damit die Anzahl seiner Schritte die Anzahl der gefundenen Körner des Sammelhamsters wiederspiegelt. Denn der Zählhamster dreht sich um und läuft zurücl, wenn er auf eine Mauer trifft; dies würde das Ergebnis insofern verfälschen, als dass wir mit der Länge der Strecke, die der Zählhamster zurücklegt, die Anzahl der Körner darstellen wollen, die der Sammelhamster aufgenommen hat.

3.1.3 Lösung A: Vorgehensweise vieler Anfänger

Die Überschrift soll einen kleinschrittigen Lösungsweg andeuten, bei dem Zeile für Zeile vorgegangen wird und immer nur das kleine Teilproblem bedacht wird, dass z.B. mit der gerade zu implementierenden Methode gelöst werden soll.
Zuerst implementieren wir eine Klasse `SammelHamster`. Der Sammelhamster kann nichts anderes tun, als sich per Zufall für eine Richtung zu entscheiden, einen Schritt vorzugehen (wenn dort keine Mauer ist) und ein Korn aufzunehmen (falls eins da ist):

```
package entwurfsmuster.beobachter.beobachterA;

import entwurfsmuster.AllroundHamster;

public class SammelHamster extends AllroundHamster {

    public SammelHamster(int r, int s, int b, int k) {
        super (r, s, b, k);
    }

    public void sammle() {
        int zufall = (int) (Math.random() * 4);
        switch (zufall) {
        case 0:
            vorUndNimm();
            break;
        case 1:
```

```
            linksUm();
            vorUndNimm();
            break;
        case 2:
            rechtsUm();
            vorUndNimm();
            break;
        case 3:
            kehrt();
            vorUndNimm();
            break;
        }
    }
}
```

Wir stellen fest, dass der Sammelhamster nichts von einem Zählhamster weiß, diesen also auch nicht benachrichtigen kann, wenn er ein Korn aufgenommen hat.
Der Zählhamster muss also selbstständig den Sammelhamster finden und feststellen, ob er ein Korn aufgenommen hat. Dazu muss die Klasse `ZaehlHamster` Folgendes enthalten:

- Ein Instanzattribut `koernerZahl`, das im Konstruktor mit 0 initialisiert wird. Es soll die Anzahl der Koerner des Sammelhamsters speichern.
- Eine Methode `findeSammelHamster`, die den Sammelhamster im Territorium findet und ihn (durch explizite Typumwandlung) als Instanz der Klasse SammelHamster zurückgibt.
- Eine Methode `zaehle`, die `findeSammelHamster` aufruft und die Anzahl der Körner des Sammelhamsters mit dem Wert vergleicht, der in `koernerZahl` gespeichert ist. Gibt es einen Unterschied, weiß der Zählhamster, dass der Sammelhamster ein Korn aufgenommen hat und geht einen Schritt vor (falls er vor einer Mauer steht, dreht er sich um), außerdem passt er `koernerZahl` an die Anzahl Körner an, die der Sammelhamster bei sich hat.

```
package entwurfsmuster.beobachter.beobachterA;

import entwurfsmuster.AllroundHamster;

public class ZaehlHamster extends AllroundHamster {

    int koernerZahl;

    public ZaehlHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
        this.koernerZahl = 0;
    }

    public SammelHamster findeSammelHamster() {
        Hamster[] alleHamster = Territorium.getHamster();
        for (Hamster hamster : alleHamster) {
```

```
            if (hamster instanceof SammelHamster) {
                return (SammelHamster) hamster;
            }
        }
        return null;
    }

    public void zaehle() {
        SammelHamster sammelHamster = findeSammelHamster();
        if (sammelHamster != null) {
            if (sammelHamster.getAnzahlKoerner()
                    == this.koernerZahl + 1) {
                this.koernerZahl++;
                if (vornFrei()) {
                    vor();
                } else {
                    kehrt();
                    if (vornFrei()) {
                        vor();
                    }
                }
            }
        }
    }

}
```

Im Hauptprogramm schließlich müssen (nach Erzeugung der beiden Hamster) die Hamster immer abwechselnd ihre namensgebenden Tätigkeiten jeweils einmal durchführen (damit der Zählhamster auch immer gleich herausfinden kann, ob der Sammelhamster ein Korn gefunden hat). Dies wird durch eine `while`-Schleife realisiert. Die Abbruchbedingung ist erfüllt, wenn kein Korn mehr im Territorium ist.

```
import entwurfsmuster.beobachter.beobachterA.ZaehlHamster;
import entwurfsmuster.beobachter.beobachterA.SammelHamster;

void main() {
    SammelHamster sammelHamster = new SammelHamster(1, 1, Hamster.WEST, 0);
    ZaehlHamster zaehlHamster = new ZaehlHamster(3, 0, Hamster.OST, 0);

    while (Territorium.getAnzahlKoerner() != 0) {
        sammelHamster.sammle();
        zaehlHamster.zaehle();
    }
}
```

Diese Lösung ist korrekt im Sinne der Aufgabenstellung. Trotzdem hat sie einige Nachteile, die wir uns im Folgenden anschauen wollen:

- Die Kommunikation findet nicht direkt zwischen den Hamstern statt. Dem Sammelhamster ist überhaupt nicht bekannt, dass es so etwas wie den Zählhamster gibt. Und der Zählhamster muss sich umständlich beim Territorium nach der gewünschten Information durchfragen, und zwar nach jedem Schritt des Sammelhamsters(!). Dies ist zudem nur durch die `while`-Schleife im Hauptprogramm möglich.

- Wenn der Zählhamster nun auf diesem umständlichen Weg einen Sammelhamster gefunden hat, kann er nicht sicher sein, dass es „der Richtige“ ist, d.h. sobald es mehrere Sammelhamster gibt, ist es nicht ausgeschlossen, dass der Zählhamster mal auf diesen mal auf jenen Sammelhamster reagiert – in Abhängigkeit davon, wie das Array von `Territorium.getHamster` aufgebaut wird. Eine eindeutige Kopplung des Zählhamsters an *einen* Sammelhamster wäre wünschenswert.

- Da es keine Kommunikation zwischen den Hamstern gibt, kann diese auch nicht wiederverwendet werden. Die Wahrscheinlichkeit ist jedoch hoch, dass eine Lösung für dieses Problem – ein Hamster sagt einem anderen Bescheid, dass ein bestimmtes Ereignis eintritt – auch für andere Hamsteraufgaben eingesetzt werden kann. Deswegen is es wünschenswert, sie abstrakt und wiederverwendbar zu implementieren.

3.1.4 Lösung B: erste Verbesserung

Wir versuchen uns zunächst daran, auf den zweiten Kritikpunkt einzugehen: Es gibt die Möglichkeit, eine allgemeingültige ID für alle Hamster im Territorium zu vergeben, unabhängig davon, von welchen Klassen die Hamsterobjekte instanziiert sind. Dazu implementieren wir eine Klasse `EindeutigerHamster`, die mit einem Instanzattribut `identifizierung` ausgestattet wird. Das Entscheidende ist ein zusätzliches Klassenattribut `zaehler`: Im Konstruktor wird zuerst `identifizierung` der Wert von `zaehler` zugewiesen und dann `zaehler` inkrementiert. Da `zaehler` bei jedem Aufruf des Konstruktors inkrementiert wird, wird jeder Instanz der Klasse `EindeutigerHamster` (und jeder Instanz von Unterklassen von `EindeutigerHamster`!) ein anderer Wert für `identifizierung` zugewiesen, da `zaehler` ein Klassenattribut ist.[9]
Die Klassen `SammelHamster` und `ZaehlHamster` müssen also von `EindeutigerHamster` erben. Damit sie trotzdem Unterklassen von `AllroundHamster` bleiben, muss `EindeutigerHamster` von `AllroundHamster` abgeleitet werden:

```
package entwurfsmuster.beobachter.beobachterB;

import entwurfsmuster.AllroundHamster;

public class EindeutigerHamster extends AllroundHamster {

    private static int zaehler = 1;
    private int identifizierung;

    public EindeutigerHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
        this.identifizierung = zaehler;
        this.zaehler++;
```

[9] Zur Wiederholung empfehle ich [BB04], S.125 (Kapitel 6.4 Klassenattribute)

```
    }

    public int getIdentifizierung() {
        return this.identifizierung;
    }
}
```

Außerdem beginnen wir, vorausschauend zu denken: Wir implementieren in dieser Klasse zusätzlich eine Methode `getIdentifizierung`, die später in der Klasse `ZaehlHamster` aufgerufen werden kann, um Kontakt mit der zugeordneten Instanz von `SammelHamster` zu bekommen.
Die Klasse `SammelHamster` muss so verändert werden, dass sie von `EindeutigerHamster` erbt, im übrigen Code ändert sich nichts:

```
package entwurfsmuster.beobachter.beobachterB;

import entwurfsmuster.AllroundHamster;

public class SammelHamster extends EindeutigerHamster {

    public SammelHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }

    public void sammle() {...}
}
```

Die Klasse `ZaehlHamster` muss ebenfalls von `EindeutigerHamster` abgeleitet werden. Außerdem führen wir ein InstanzAttribut `idSammelHamster` ein, über das ein Zählhamster schon bei der Erzeugung mit einem Sammelhamster verknüpft wird. Dies ist noch eine sehr lose Kopplung, da der Zählhamster den Sammelhamster noch nicht direkt ansprechen kann, sondern wieder erst im Territorium nach dem richtigen suchen muss.
Nun können wir den geschaffenen Vorteil in der Methode `findeSammelHamster` nutzen und haben die Sicherheit, dass ein Zählhamster auch bei mehreren Sammelhamstern im Territorium immer „seinen“ Sammelhamster findet:

```
package entwurfsmuster.beobachter.beobachterB;

import entwurfsmuster.AllroundHamster;

public class ZaehlHamster extends EindeutigerHamster {

    int koernerZahl;
    int idSammelHamster;

    public ZaehlHamster(int r, int s, int b, int k, int idSammelHamster) {
        super(r, s, b, k);
        this.koernerZahl = 0;
        this.idSammelHamster = idSammelHamster;
```

```
    }

    public SammelHamster findeSammelHamster() {
        EindeutigerHamster[] alleHamster =
                (EindeutigerHamster[]) Territorium.getHamster();
        for (EindeutigerHamster hamster : alleHamster) {
            if (hamster.getIdentifizierung() == this.idSammelHamster) {
                return (SammelHamster) hamster;
            }
        }
        return null;
    }

    public void zaehle() {...}
}
```

Im Hauptprogramm müssen wir nur berücksichtigen, dass der Konstruktor von `ZaehlHamster` einen weiteren Parameter hat, nämlich die ID des Sammelhamsters.

```
import entwurfsmuster.beobachter.beobachterB.ZaehlHamster;
import entwurfsmuster.beobachter.beobachterB.SammelHamster;

void main() {
    SammelHamster sammelHamster = new SammelHamster(1, 1, Hamster.WEST, 0);
    int sammelHamsterId = sammelHamster.getIdentifizierung();
    ZaehlHamster zaehlHamster =
             new ZaehlHamster(3, 0, Hamster.OST, 0, sammelHamsterId);

    while (Territorium.getAnzahlKoerner() != 0) {
        sammelHamster.sammle();
        zaehlHamster.zaehle();
    }
}
```

3.1.5 Lösung C: Kommunikation

Die anderen Nachteile sind schwieriger zu beheben. Wir stellen fest, dass die übrigbleibende Kritik ausschließlich mit der fehlenden Kommunikation zwischen den Hamstern zu tun hat. Wir müssen sie also enger aneinander knüpfen – ein erstes „einander Vorstellen“ hat schon in Lösung B stattgefunden, nun müssen wir sie „miteinander bekannt machen“. Dies können wir tun durch eine einfache Technik der objektorientierten Programmierung: Wir geben der Klasse `SammelHamster` ein Instanzattribut vom Typ `ZaehlHamster`.
Dadurch ergeben sich folgende weitere Änderungen:

- Im Konstruktor muss das Attribut `zaehlHamster` initialisiert werden.
- Durch diese Bindung brauchen wir die eindeutige ID und die Hilfsklasse nicht mehr und leiten wieder direkt von `AllroundHamster` ab.
- Die Methode `vor` des Objektes, auf das `zaehlHamster` eine Referenz ist, kann nun direkt nach dem Aufruf der Methode `nimm` beim `SammelHamster` aufgerufen werden. Dadurch

wiederum kann nicht mehr die Methode `vorUndNimm` der Oberklasse `AllroundHamster` verwendet und muss entsprechend überschrieben werden.

- Da der Sammelhamster sich nun darum kümmert, dass der Zählhamster immer einen Schritt vorgeht, kann die `while`-Schleife aus dem Hauptprogramm in seine Methode `sammle` verlagert werden.

```
package entwurfsmuster.beobachter.beobachterC;

import entwurfsmuster.AllroundHamster;

public class SammelHamster extends AllroundHamster {

    ZaehlHamster zaehlHamster;

    public SammelHamster(int r, int s, int b, int k,
                         ZaehlHamster zaehlHamster) {
        super(r, s, b, k);
        this.zaehlHamster = zaehlHamster;
    }

    public void vorUndNimm () {
        if (vornFrei()) {
            vor();
            if (kornDa()) {
                nimm();
                if (this.zaehlHamster.vornFrei()) {
                    this.zaehlHamster.vor();
                } else {
                    this.zaehlHamster.kehrt();
                    if (this.zaehlHamster.vornFrei()) {
                        this.zaehlHamster.vor();
                    }
                }
            }
        }
    }

    public void sammle() {
        while (Territorium.getAnzahlKoerner() != 0 ) {
            int zufall =(int) (Math.random() * 4);
            switch (zufall) {...}
        }
    }
}
```

Die Klasse `ZaehlHamster` kann nun stark vereinfacht werden:

- Sie muss nicht mehr von `EindeutigerHamster` erben.

- Das Attribut `koernerZahl` entfällt.
- Die Methoden `findeSammelHamster` und `zaehle` werden nicht mehr benötigt

Mit anderen Worten: Im Prinzip benötigen wir die Klasse `ZaehlHamster` nicht mehr, wir könnten für unser Programm eine Instanz der Klasse `AllroundHamster` verwenden. Der Verständlichkeit halber behalten wir jedoch die Klasse `ZaehlHamster` bei, auch wenn sie bei dieser Lösungsvariante sehr minimal ausfällt:

```
package entwurfsmuster.beobachter.beobachterC;

import entwurfsmuster.AllroundHamster;

public class ZaehlHamster extends AllroundHamster {

    public ZaehlHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }
}
```

Verbleibt noch das leicht veränderte Hauptprogramm, in der die `while`-Schleife durch einen einfachen Aufruf der Methode `sammle` der `SammelHamster`-Instanz ersetzt wird. Außerdem wird der Sammelhamster direkt mit „seinem“ Zählhamster initialisiert. Wir haben also zum einen die Änderung, dass nun der Sammelhamster den Zählhamster kennt, dieser aber den Sammelhamster nicht mehr; und zum anderen einen noch nicht angesprochenen Vorteil: Es kann nicht passieren, dass der Sammelhamster anfängt zu sammeln und bei dem Finden eines Korns feststellt, dass es gar keinen Zählhamster gibt.

```
import entwurfsmuster.beobachter.beobachterC.ZaehlHamster;
import entwurfsmuster.beobachter.beobachterC.SammelHamster;

void main() {
    ZaehlHamster zaehlHamster = new ZaehlHamster(3, 0, Hamster.OST, 0);
    SammelHamster sammelHamster =
            new SammelHamster(1, 1, Hamster.WEST, 0, zaehlHamster);

    sammelHamster.sammle();
}
```

3.1.6 Lösung D: Langfristig denken

Die eben vorgestellte ist wohl die kürzeste Lösung. Allerdings legt uns der Vorteil, die Methode `vor` der Klasse `ZaehlHamster` direkt in der Klasse `SammelHamster` aufzurufen, auf den zweiten Blick zu sehr fest, da der Zählhamster nicht entscheiden kann, was er tut. Besser wäre es, wenn der Sammelhamster dem Zählhamster nur Bescheid gäbe, dass das besagte Ereignis (ein Korn wurde eingesammelt) eingetreten ist, und der nun selber entscheidet, was er tut. In unserem Fall soll er einen Schritt vor gehen, aber er könnte ja auch etwas anderes tun. Wenn man hier variieren will, ist es für die Übersichtlichkeit besser, die Änderung im ZaehlHamster

vornehmen zu können. Außerdem kann man auch der Meinung sein, dass es die Kapselung des Zählhamsters verletzt, wenn der Sammelhamster einfach bestimmen kann, was jener tut.[10] Dazu geben wir der Klasse `ZaehlHamster` eine Methode `aktualisiere`, in der die gewünschte Reaktion des Hamsters auf ein Ereignis implementiert wird:

```
package entwurfsmuster.beobachter.beobachterD;

import entwurfsmuster.AllroundHamster;

public class ZaehlHamster extends AllroundHamster {

    public ZaehlHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }

    public void aktualisiere() {
        if (vornFrei()) {
            vor();
        } else {
            kehrt();
            if (vornFrei()) {
                vor();
            }
        }
    }
}
```

Im `SammelHamster` wird nun diese neue Methode statt `vor` aufgerufen. Damit gibt dieser die Kontrolle darüber ab, was der Zählhamster genau tut, wenn er ein Korn aufgenommen hat – wir können auch sagen: Er weiß nichts mehr darüber.

```
package entwurfsmuster.beobachter.beobachterD;

import entwurfsmuster.AllroundHamster;

public class SammelHamster extends AllroundHamster {

    private ZaehlHamster zaehlHamster;

    public SammelHamster(int r, int s, int b, int k,
                         ZaehlHamster zaehlHamster) {
        super(r, s, b, k);
        this.zaehlHamster = zaehlHamster;
    }

    public void vorUndNimm() {
        if (vornFrei()) {
```

[10] Zum Begriff der Kapselung bzw. Verkapselung siehe [Bal01], S.821

```
            vor();
        }
        if (kornDa()) {
            nimm();
            this.zaehlHamster.aktualisiere();
        }
    }

    public void sammle() {...}
}
```

Das Hauptprogramm bleibt unverändert.

3.1.7 Lösung E: Das Beobachtermuster

Bis auf die Wiederverwendbarkeit sind nun alle Kritikpunkte ausgeräumt. Diese können wir durch das Beobachtermuster erreichen. Im Grunde arbeiten wir schon fast damit, wir müssen nur noch die „Empfänger"-Funktionalität von der Klasse `ZaehlHamster` trennen (indem wir ihn von einem Hamster ableiten, der nur besagte Funktionalität implementiert) und die „Sender"-Funktionalität vom `SammelHamster` (was wir ebenso machen).
Das Interface `EmpfaengerHamster` tut nichts anderes, als jedem Hamster, der es implementiert, vorzuschreiben, eine Methode `aktualisiere` zu deklarieren.

```
package entwurfsmuster.beobachter.beobachterE;

public interface EmpfaengerHamster {
    public void aktualisiere();
}
```

Alles andere erledigt die Klasse `SendeHamster`. Um die Wiederverwendbarkeit weiter zu erhöhen, kann eine `SendeHamster`-Instanz nicht nur mit einer, sondern mit mehreren Instanzen von `EmpfaengerHamster` kommunizieren. Dazu verwaltet sie in einer ArrayList[11] alle `EmpfaengerHamster`, an die sie sendet. Mit den Methoden `meldeAn` und `meldeAb` können sich `EmpfaengerHamster` in die Liste ein- und aus der Liste austragen (dazu müssen sie mit dem Zugriffsrechtstyp `public` sein). Mit der Methode `benachrichtige` kann schließlich ein `SendeHamster` alle eingetragenen `EmpfaengerHamster` benachrichtigen, dass ein bestimmtes Ereignis eingetreten ist.

```
package entwurfsmuster.beobachter.beobachterE;

import java.util.List;
import java.util.ArrayList;
import entwurfsmuster.AllroundHamster;

abstract public class SendeHamster extends AllroundHamster {

    List<EmpfaengerHamster> alleEmpfaenger;
```

[11] Siehe Kapitel 2.4.1 ArrayList S.14

```
    public SendeHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
        this.alleEmpfaenger = new ArrayList<EmpfaengerHamster>();
    }

    public void meldeAn(EmpfaengerHamster empfaengerHamster) {
        this.alleEmpfaenger.add(empfaengerHamster);
    }

    public void meldeAb(EmpfaengerHamster empfaengerHamster) {
        this.alleEmpfaenger.remove(empfaengerHamster);
    }

    protected void benachrichtige() {
        for (EmpfaengerHamster hamster : this.alleEmpfaenger) {
            hamster.aktualisiere();
        }
    }
}
```

Zusammengefasst haben wir also folgende Neuerungen:

- Ein Sendehamster kann mit mehreren Empfängerhamstern kommunizieren.
- Empfängerhamster können sich während des Programmablaufs beim Sendehamster an- und abmelden, d.h. die Kopplung ist flexibler.
- Die Klasse `SendeHamster` stellt die Methode `benachrichtige` zwar bereit, ruft sie aber selbst gar nicht auf. Dies ist so gewollt, denn von ihr selbst sollen ja gar keine Objekte instanziiert werden (deswegen ist er abstrakt). Unterklassen, die von `SendeHamster` erben, benutzen die hier bereitgestellten Methoden für ihren jeweiligen Zweck. Wir können ihn also für unsere `SammelHamster`-Klasse verwenden, wir könnten aber auch etwas anderes damit tun, haben also eine größere Wiederverwendbarkeit erreicht.

Nun leiten wir als erstes `SammelHamster` von `SendeHamster` ab. Hier haben wir nun eine Vereinfachung: Das Instanzattribut `zaehlHamster` können wir weglassen, da die Speicherung und Verwaltung der jeweiligen `ZaehlHamster`-Instanzen komplett von den geerbten Methoden der Oberklasse übernommen wird.
Statt direkt die `aktualisiere`-Methode *eines* `EmpfaengerHamster`-Objektes aufzurufen, rufen wir nun die `benachrichtige`-Methode auf, welche die `aktualisiere`-Methoden von *allen* angemeldeten `EmpfaengerHamster`-Objekten aufruft.

```
package entwurfsmuster.beobachter.beobachterE;

public class SammelHamster extends SendeHamster {

    public SammelHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }
```

```
    public void vorUndNimm() {
        if (vornFrei()) {
            vor();
        }
        if (kornDa()) {
            nimm();
            benachrichtige();
        }
    }

    public void sammle() {...}
}
```

Die `ZaehlHamster`-Klasse bleibt unverändert, außer, dass sie nun `EmpfaengerHamster` implementiert. Dieses Interface benötigen wir im Moment eigentlich nicht, da bei der Lösung unserer Aufgabe die Menge der `ZaehlHamster`-Objekte mit der Menge der `EmpfaengerHamster`-Objekte identisch sein dürfte.
Da wir aber mit Weitsicht programmieren wollen, behalten wir es bei. Denn in anderen Fällen könnte es so sein, dass verschiedene Klassen `EmpfaengerHamster` sein sollen und für diesen Fall ist das Interface unverzichtbar, da man es als gemeinsame Schnittstelle benötigt, um alle `EmpfaengerHamster` ansprechen zu können.
Generell empfehle ich an dieser Stelle zum tieferen Verständnis eine Wiederholung der Themen *Abstrakte Klassen* und *Interfaces*.[12]

```
package entwurfsmuster.beobachter.beobachterE;

import entwurfsmuster.AllroundHamster;

public class ZaehlHamster extends AllroundHamster
                          implements EmpfaengerHamster {

    public ZaehlHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }

    public void aktualisiere() {
        if (vornFrei()) {
            vor();
        } else {
            kehrt();
            if (vornFrei()) {
                vor();
            }
        }
    }
}
```

[12] [BB04] S.333 (Kapitel 12 Abstrakte Klassen und Interfaces)

Das neue Hauptprogramm ähnelt sehr dem alten, nur dass das `SammelHamster`-Objekt nicht mehr mit dem `ZaehlHamster`-Objekt initialisiert wird, sondern letzteres meldet sich bei ersterem an:

```
import entwurfsmuster.beobachter.beobachterE.ZaehlHamster;
import entwurfsmuster.beobachter.beobachterE.SammelHamster;

void main() {
    ZaehlHamster zaehlHamster = new ZaehlHamster(3, 0, Hamster.OST, 0);
    SammelHamster sammelHamster = new SammelHamster(1, 1, Hamster.WEST, 0);

    sammelHamster.meldeAn(zaehlHamster);
    sammelHamster.sammle();
}
```

Was bringt uns diese Vorgehensweise nun für Vorteile ein?

- Die Kommunikation ist unabhängig von dem implementiert, was die Hamster sonst tun, d.h. wir können das Interface `EmpfaengerHamster` und die Klasse `SendeHamster` immer verwenden, wenn wir in beliebiger Form einen Hamster benötigen, der 1 bis n andere Hamster über ein bestimmtes Ereignis informieren soll, damit diese wiederum entsprechend reagieren können.
- Die Kopplung zwischen `EmpfaengerHamstern` und `SendeHamstern` ermöglicht direkte Kommunikation, ist aber auf ein Minimum reduziert:
    1. Ein `SendeHamster` weiß nur, dass auf seiner Liste `EmpfaengerHamster` stehen, aber was diese sonst noch tun, weiß er nicht.
    2. Zur Laufzeit können sich `BeobachterHamster` bei verschiedenen `SendeHamstern` an- und abmelden, sind also nicht darauf reduziert, das schon bei der Initialisierung entscheiden zu müssen.
    3. Ein `BeobachterHamster` weiß nichts von anderen `BeobachterHamstern`.

3.1.8 Zusammenfassung und Ausblick

Das Entwurfsmuster Beobachter ist ein objektbasiertes Verhaltensmuster. Abbildung 2 zeigt ein Klassendiagramm, das abstrakt die wesentlichen Merkmale des Musters zusammenfasst. Bei diesem sehr ausführlichen Beispiel wurde deutlich, wie Software-Entwicklung oft abläuft: Man plant aufgrund des aktuellen Kenntnisstandes, implementiert und stellt fest, dass sich Nachteile ergeben. Aufgrund dieser neuen Kenntnisse kann dann ein neuer Planungsschritt erfolgen. Dies scheint ein guter Mittelweg zu sein zwischen der strikten Vorgabe, zuerst alles bis ins kleinste Detail zu planen und alle Konsequenzen vorauszuahnen (was in der Praxis meist unrealistisch ist) und dem immer noch häufigen kurzsichtigen Programmieren, dass nur kleine lokale Probleme Zeile für Zeile lösen kann.
Es folgen einige Hinweise, die bei der vertiefenden Beschäftigung mit diesem Muster nützlich seien könnten:

1. *Verschiedene Namen* In [GHJV04] ist nicht von „Sendern“ und „Empfängern“, sondern von „Subjekten“ und „Beobachtern“ die Rede. In [BMRSM00] ist dieses Entwurfsmuster unter dem Namen „Publisher-Subscriber“ beschrieben.

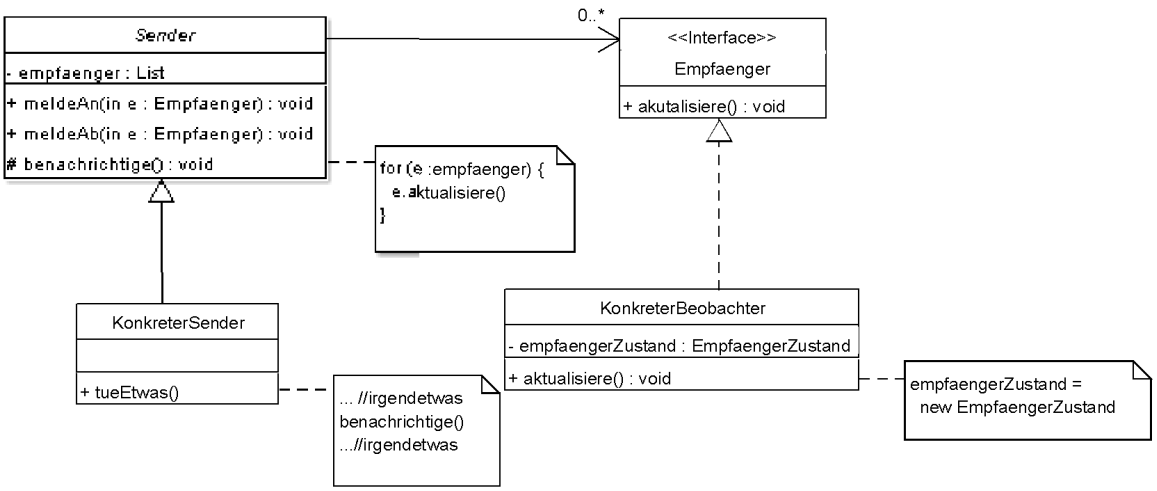


Abbildung 2: Klassendiagramm zum Entwurfsmuster Beobachter

2. *Nachteil der losen Kopplung* In einem komplexen System kann die Übersicht darüber verloren werden, welche Empfänger sich zur Laufzeit bei einem Sender anmelden und vielleicht angepasst werden müssen, wenn der Sender geändert wird.

3. *Verteilung von Informationen* Das Beobachtermuster legt nicht fest, wieviel Information der Sender dem Empfänger zu kommen lässt. Dies muss kontextspezifisch entschieden werden. Die beiden Extreme haben besondere Namen: Wenn ein Sender nur mitteilt, *dass* sich etwas geändert hat, aber nichts darüber, *was* sich geändert hat, so sprechen wir vom *Pull-Modell*. Wenn der Sender alle Informationen über diese Änderung mitteilt, die er selber hat – gleichgültig, ob der Empfänger die Information braucht oder nicht –, so sprechen wir vom *Push-Modell*.

   Ein Kompromiss zwischen diesen beiden Extremen ist die Anwendung eines *Aspekt-Konzepts*, bei der ein Empfänger dem Sender bei der Anmeldung mitteilt, über welchen Aspekt einer Änderung er informiert werden möchte.

4. *Änderungsmanager* Wenn komplexe Aktualisierungen notwendig sind, kann ein Änderungsmanager von Vorteil sein. Dies ist eine spezielle Klasse, in der festgehalten wird, welche Empfänger von welchen Sendern welche Informationen haben möchten. Der Vorteil liegt hier darin, dass die entsprechenden Wünsche dem Änderungsmanager nur einmal mitgeteilt werden müssen, die Empfänger und Sender selbst müssen sich aber danach nicht mehr darum kümmern.

   Der Änderungsmanager ist ein schönes Beispiel für das Zusammenarbeiten mehrere Entwurfsmuster, da diese Klasse typischerweise ein Vermittler (siehe Kapitel 3.6, Seite 71) und ein Singleton (siehe Kapitel 3.2, Seite 37) ist.

5. *Variationen:* Das hier vorgestellte Hamsterprogramm verwendet eine von vielen möglichen Implementierungsmöglichkeiten des Beobachter-Musters. Andere Möglichkeiten sind zum Beispiel:

   - Ein Objekt ist gleichzeitig Sender und Empfänger.
   - Ein Empfänger merkt sich den Sender, bei dem er angemeldet ist, um abzusichern, dass er nur von diesem aktualisiert wird.

- Ein Empfänger meldet sich bei mehreren Sendern an und reagiert auf Benachrichtigungen von unterschiedlichen Sendern mit unterschiedlichen Aktualisierungen.

3.2 Singleton

Das Singleton-Muster stellt sicher, dass nur ein Objekt von einer Klasse instanziiert werden kann; es wurde bereits in [BB04], S.455 vorgestellt. Deswegen möchte ich an dieser Stelle nur kurz ein weiteres Beispiel geben, das zeigt, wie das Muster sinnvoll genutzt werden kann:
Immer, wenn Hamster ein Spiel miteinander spielen, ist es praktisch, einen weiteren Hamster zu haben, der die Regeln überwacht – oder abstrakt gesprochen: Wenn mehrere Objekte komplex miteinander interagieren, ist ein Objekt hilfreich, das „den Überblick“ hat und die Interaktionen der anderen überwacht, da die immer nur ihre lokale Situation kennen. Was man vermeiden will, sind mehrere solcher übergeordneten Objekte bzw. Hamster, die miteinander konkurrieren, da dies zu unvorhergesehenen Ereignissen führen könnte.

3.2.1 Motivationsaufgabe: Ein Wettrennen

Wir möchten unsere beiden Hamster aus der letzten Aufgabe verwenden, um ein Hamsterwettrennen zu implementieren. Genauer: Zählhamster sollen miteinander um die Wette laufen.
Die Zählhamster starten nebeneinander – also zum Beispiel in derselben Spalte in aufeinanderfolgenden Reihen – und das Rennen ist zu Ende, wenn der erste von ihnen die Ziellinie überschreitet bzw. eine „Zielspalte“ betritt.
Jeder Zählhamster ist über das Beobachtermuster mit einem Sammelhamster verknüpft. Die Sammelhamster starten auf demselben Feld. Sie dürfen immer abwechselnd einen Zug tun und – wie gehabt – wenn sie ein Korn aufnehmen, geht der zugeordnete Zählhamster einen Schritt vor.
Dazu implementieren wir noch einen Schiedsrichterhamster, der die Spalte überwacht, die als Ziellinie dienen soll. Zusätzlich ist er dafür zuständig, den Sammelhamstern immer abwechselnd jeweils einen Schritt zu erlauben. Diesen Hamster machen wir zum Singleton, damit wirklich nur einer über den Ablauf bestimmen und den Sieger küren kann.
Übrigens kann dieses „Wettrennen“ sehr gut zum Test von verschiedenen Sammelalgorithmen dienen. Hat man diese nicht zur Hand, genügen vier gleichartige Sammelhamster. Welcher von ihnen gewinnt, ist dann vollständig dem Zufall überlassen.

3.2.2 Territorium

Das Territorium sollte groß genug sein, um den Zaehlhamstern ihr Wettrennen zu ermöglichen und genug Körner enthalten, damit die Sammelhamster welche finden können und das Rennen zügig vorangeht. Zudem sollte es keine Mauern innerhalb des Territoriums geben.

3.2.3 Lösung A: Der Schiedsrichter

Wenden wir uns nun der konkreten Implementierung der Klasse `SchiedsrichterHamster` zu – und zwar zunächst ohne angewandtes Singleton-Muster:

- Wir legen in der Konstanten `SPIELER_ANZAHL_MAX` fest, wieviele Spielerpaare wir maximal zulassen wollen.
- In zwei Attributen vom Typ `ArrayList` werden die Zählhamster und Sammelhamster gespeichert, die sich zum Spiel anmelden; wir werden sie im Folgenden als Läufer und Sammler bezeichnen.

- Mit der Methode `zumSpielAnmelden` können sich Hamster in diese Listen eintragen und sie werden von ihr korrekt einsortiert. Allerdings nur, wenn sie Instanzen von `ZaehlHamster` oder `SammelHamster` sind und die maximale Anzahl an Spielern noch nicht erreicht ist.

- Die Methode `spielAblauf` ist für Folgendes zuständig: Sie ordnet die Läufer den Sammlern zu, d.h. sie meldet die `ZaehlHamster`-Objekte bei den `SammelHamster`-Objekten an. Bis ein Hamsterpaar gewonnen hat, dürfen alle Sammler in der Reihenfolge einen Schritt tun, dann wird kontrolliert, ob damit der zugehörige Läufer die Ziellinie überschritten hat und wenn das der Fall ist, wird das Gewinnerpaar bekannt gegeben.

  Dazu werden drei Hilfsmethoden verwendet, die als `private` deklariert sind (weil sie nur intern benutzt werden sollen) und nach `spielAblauf` implementiert werden – in der Reihenfolge, wie sie in `spielAblauf` verwendet werden.

```
package entwurfsmuster.singleton.singletonA;

import java.util.List;
import java.util.ArrayList;
import entwurfsmuster.beobachter.beobachterE.ZaehlHamster;

public class SchiedsrichterHamster extends Hamster {

    public static final int SPIELER_ANZAHL_MAX = 4;
    private List<ZaehlHamster> laeufer;
    private List<SammelHamster> sammler;

    public SchiedsrichterHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
        this.laeufer = new ArrayList<ZaehlHamster>();
        this.sammler = new ArrayList<SammelHamster>();
    }

    public void meldeZumSpielAn(Hamster hamster) {
        if (hamster instanceof ZaehlHamster
                && this.laeufer.size() <= SPIELER_ANZAHL_MAX) {
            this.laeufer.add((ZaehlHamster)hamster);
        } else if (hamster instanceof SammelHamster
                        && this.sammler.size() <= SPIELER_ANZAHL_MAX) {
            this.sammler.add((SammelHamster)hamster);
        } else {
            schreib("Anmeldung nicht möglich.");
        }
    }

    public void steuereAblauf() {
        boolean gewonnen = false;
        ordneHamsterZu();
        while (!gewonnen) {
```

```
            for (int i = 0; i < laeufer.size(); i++) {
                ZaehlHamster zaehlHamster = this.laeufer.get(i);
                SammelHamster sammelHamster = this.sammler.get(i);
                sammelHamster.sammle();
                gewonnen = kontrolliereSpalten(zaehlHamster);
                if (gewonnen) {
                    gebeGewinnerBekannt(zaehlHamster, sammelHamster);
                    return;
                }
            }
        }
    }

    private void ordneHamsterZu() {
        if (this.laeufer.size() == this.sammler.size()) {
            for (int i=0; i < laeufer.size(); i++) {
                sammler.get(i).meldeAn(laeufer.get(i));
            }
        }
    }

    private boolean kontrolliereSpalten(ZaehlHamster zaehlHamster) {
        return this.getSpalte() == zaehlHamster.getSpalte();
    }

    private void gebeGewinnerBekannt(ZaehlHamster laeufer,
                                     SammelHamster sammler) {
        schreib("Gewonnen haben " + laeufer.toString() + " und "
                    + sammler.toString() + "!");
    }
}
```

Damit das so funktionieren kann, müssen wir auch den `SammelHamster` leicht anpassen: Wir müssen die `while`-Schleife aus der Methode `sammeln` entfernen.

```
package entwurfsmuster.singleton.singletonA;

import entwurfsmuster.beobachter.beobachterE.SendeHamster;

public class SammelHamster extends SendeHamster {

    public SammelHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }

    public void vorUndNimm() {
        if (vornFrei()) {
            vor();
            if (kornDa()) {
```

```
                nimm();
                benachrichtige();
            }
        }
    }

    public void sammle() {
        int zufall = (int) (Math.random() * 4);
        switch (zufall) {...}
    }
}
```

Im Hauptprogramm werden nun alle Hamster (vier Läufer-Sammler-Paare und der Schiedsrichter) erzeugt, dann alle Spieler beim Schiedsrichter angemeldet und schließlich der Spielablauf ausgeführt.

```
import entwurfsmuster.beobachter.beobachterE.ZaehlHamster;
import entwurfsmuster.singleton.singletonA.SammelHamster;
import entwurfsmuster.singleton.singletonA.SchiedsrichterHamster;

void main() {
    SchiedsrichterHamster schiedsRichter =
            new SchiedsrichterHamster(2, 5, Hamster.SUED, 0);
    SammelHamster sammler1 = new SammelHamster(10, 10, Hamster.OST, 0);
    SammelHamster sammler2 = new SammelHamster(10, 10, Hamster.OST, 0);
    SammelHamster sammler3 = new SammelHamster(10, 10, Hamster.OST, 0);
    SammelHamster sammler4 = new SammelHamster(10, 10, Hamster.OST, 0);
    ZaehlHamster laeufer1 = new ZaehlHamster(3, 0, Hamster.OST, 0);
    ZaehlHamster laeufer2 = new ZaehlHamster(4, 0, Hamster.OST, 0);
    ZaehlHamster laeufer3 = new ZaehlHamster(5, 0, Hamster.OST, 0);
    ZaehlHamster laeufer4 = new ZaehlHamster(6, 0, Hamster.OST, 0);

    schiedsRichter.meldeZumSpielAn(laeufer1);
    schiedsRichter.meldeZumSpielAn(laeufer2);
    schiedsRichter.meldeZumSpielAn(laeufer3);
    schiedsRichter.meldeZumSpielAn(laeufer4);
    schiedsRichter.meldeZumSpielAn(sammler1);
    schiedsRichter.meldeZumSpielAn(sammler2);
    schiedsRichter.meldeZumSpielAn(sammler3);
    schiedsRichter.meldeZumSpielAn(sammler4);

    schiedsRichter.steuereAblauf();
}
```

3.2.4 Lösung B: Schiedsrichter als Singleton

Stellen wir uns nun vor, es werden versehentlich zwei Schiedsrichterhamster erzeugt, und die Hamster melden sich bei dem einen an und der andere erzeugt den Spielablauf. Ein kurzer Blick auf die Implementierung zeigt uns, dass damit das Spiel nicht mehr funktionieren würde.

Wie bei all unseren Beispielen gilt hier wieder: Natürlich ist es unwahrscheinlich, dass so etwas bei dieser geringen Programmkomplexität aus Versehen passieren kann – doch üben wir ja für den „Ernstfall“ mit einer weit größeren Programmkomplexität. Deswegen machen wir nun aus dem Schiedsrichter ein Singleton.
Um eine gewöhnliche Klasse zu einer Singleton-Klasse zu machen, müssen wir Folgendes tun:

1. den Konstruktor vor äußeren Zugriffen schützen (mit dem Zugriffsrechtstyp `private`)
2. ein (wieder mit `private`) geschütztes Klassenattribut mit der Klasse selbst als Typ implementieren
3. eine Klassenmethode `getExemplar` zur Verfügung stellen, die das Objekt, welches von diesem Klassenattribut referenziert wird, nach außen gibt

Angewandt auf unsere Klasse `SchiedsrichterHamster` bedeutet das:

```
package entwurfsmuster.singleton.singletonB;

import java.util.List;
import java.util.ArrayList;
import entwurfsmuster.beobachter.beobachterE.ZaehlHamster;
import entwurfsmuster.singleton.singletonA.SammelHamster;

public class SchiedsrichterHamster extends Hamster {

    public static final int SPIELER_ANZAHL_MAX = 4;
    private static SchiedsrichterHamster exemplar;
    private List<ZaehlHamster> laeufer;
    private List<SammelHamster> sammler;

    public static SchiedsrichterHamster getExemplar(int r, int s,
                                                    int b, int k) {
        if (exemplar == null) {
            return new SchiedsrichterHamster(r, s, b, k);
        } else {
          return exemplar;
        }
    }

    private SchiedsrichterHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
        this.laeufer = new ArrayList<ZaehlHamster>();
        this.sammler = new ArrayList<SammelHamster>();
    }

    public void meldeZumSpielAn(Hamster hamster) {...}
    public void steuereAblauf() {...}
    private void ordneHamsterZu() {...}
    private boolean kontrolliereSpalten(ZaehlHamster zaehlHamster) {...}
    private void gebeGewinnerBekannt(ZaehlHamster laeufer,
```

```
                                        SammelHamster sammler) {...}
}
```

Im Hauptprogramm müssen wir nur die Zeile

```
SchiedsrichterHamster schiedsrichter =
        new SchiedsrichterHamster(2, 5, Hamster.SUED, 0);
```

durch die Zeile

```
SchiedsrichterHamster schiedsRichter =
        SchiedsrichterHamster.getExemplar(2, 5, Hamster.SUED, 0);
```

ersetzen.
Damit haben wir abgesichert, dass es nur einen SchiedsrichterHamster geben kann. Würden wir nach der Änderung der Schiedsrichterklasse weiterhin die erste Zeile verwenden, bekämen wir folgende Fehlermeldung vom Compiler:
„SchiedsrichterHamster(int,int,int,int) has private access in
entwurfsmuster.singleton.singletonB.SchiedsrichterHamster“
Zu beachten ist noch, dass die Parameter, die beim Aufruf der Methode `getExemplar` übergeben werden, nur beim ersten Aufruf verwendet werden. Danach kann man sie natürlich noch angeben, sie haben aber keine Wirkung. Dieser Umstand ist immer zu beachten, wenn man eine Klasse zum Singleton macht, für die im Konstruktor Parameter angegeben werden müssen.

3.2.5 Zusammenfassung und Ausblick

Das Entwurfsmuster Singleton ist ein objektbasiertes Erzeugermuster. Da dieses Muster nur eine einzelne Klasse betrifft, ist es – im Gegensatz zu vielen anderen Entwurfsmustern – relativ einfach, es auch nachträglich in eine bestehende Struktur einzubauen.
Ein Grund für seine Einfachheit liegt darin, dass ein als `private` deklarierter Konstruktor nicht nur verhindert, dass Objekte von der Klasse instanziiert werden können, sondern auch, dass von der Klasse abgeleitet werden kann.
In dem hier vorgestellten Hamsterprogramm ist diese Verfahrensweise gewählt worden, so muss aber nicht zwangsläufig verfahren werden. Zum Beispiel ist das Singleton-Muster in [GHJV04] mit einem `protected`-Konstruktor beschrieben.
Ein wesentliches Thema bezüglich den Einsatz von Singleton-Klassen in der Praxis ist die Vererbung. Dies lässt die Vererbung zu, bringt aber folgende Probleme mit sich:

- Innerhalb des Paketes der Singleton-Klasse kann der Konstruktor verwendet und damit das Singleton-Prinzip verletzt werden.
- Es ist nicht möglich, in der Singleton-Klasse festzulegen, dass die Unterklassen auch Singletons sein sollen.

Im Folgenden wird eine Lösungsidee für beide Probleme vorgestellt, die zwar nicht vollständig zufriedenstellend ist, aber eine brauchbare Alternative sein kann, wenn denn die Ableitung einer Singleton-Klasse unbedingt erforderlich ist:

- Die Singleton-Oberklasse verwaltet ihr eigenes und die Exemplare ihrer Unterklassen in einer `HashMap` die Klassennamen auf konkrete Objekte abbildet.

- Im Konstruktor wird überprüft, ob ein Objekt der zu instanziierenden Klasse bereits in der `HashMap` vorhanden ist. Wenn dies nicht der Fall ist, wird die Objekterzeugung zugelassen und das Objekt in der `HashMap` gespeichert. Andernfalls wird eine `RuntimeException` geworfen, um die Objekterzeugung zu verhindern.
- Die Methode `getExemplar` arbeitet vom Prinzip wie die im vorgestellten Hamsterprogramm: Wenn ein Exemplar der gewünschten Klasse zur Verfügung steht, wird dies zurückgegeben, wenn nicht, wird ein neues erzeugt und gespeichert.
- In einer Unterklasse kann eine eigene `getExemplar`-Methode angelegt werden, die ein entsprechendes Objekt von Typ der Unterklasse zurückgibt.

Der Quellcode einer solchen modifizierten Singleton-Klasse sieht wie folgt aus:

```
import java.util.Map;
import java.util.HashMap;

public class Singleton {

    protected static Map<String,Singleton> exemplare =
            new HashMap<String,Singleton>();

    protected Singleton() {
        String name = this.getClass().getName();
        if (Singleton.exemplare.get(name) != null) {
            throw new RuntimeException();
        }
        Singleton.exemplare.put(name, this);
    }

    public static Singleton getExemplar(String name) {
        if (Singleton.exemplare.get(name) != null) {
            return Singleton.exemplare.get(name);
        }
        return new Singleton();
    }
}
```

Grundsätzlich ist damit die Anforderung nach einer Klasse, die garantiert, dass es von ihr selbst sowie von all ihren Unterklassen nur ein Exemplar gibt, erfüllt.
Der Nachteil dieser Lösung ist jedoch, dass Objekterzeugungen erst zur Laufzeit verhindert werden. Wenn ein `private`-Konstruktor verwendet wird, beschwert sich im Gegensatz dazu bereits der Compiler, wie wir schon gesehen haben. Ein Programmierer, der diese Klasse benutzt, muss aus diesem Grunde vorsichtig mit dem Konstruktor umgehen, d.h. entweder eine Ausnahmebehandlung durchführen oder sicherstellen, dass der Konstruktor höchstens einmal aufgerufen wird.

3.3 Adapter

3.3.1 Motivationsaufgabe

1. Es gibt eine Gruppe von Hamstern, die tanzen. Wie sie das tun, bleibt dem Programmierer überlassen, Vorgabe ist nur, dass sie sich bewegen sollen, nicht aufeinander treten sollen und dass sie in irgendeiner Weise aufeinander eingehen sollen.

2. Es gibt eine Gruppe weiterer Hamster, die rechnen können. Wie vielfältig, schnell usw. sie das können, spielt für die Aufgabe keine Rolle, deswegen wollen wir uns mit einer möglichst simplen Funktion begnügen.

3. Es soll einen weiteren Hamster geben, der zunächst zu keiner der beiden Gruppen gehört: Er möchte gerne zu den Tanzhamstern gehören, denn er fühlt sich einsam und möchte zu einer Gemeinschaft gehören. Gleichzeitig bewundert er die Rechenhamster für ihr Können und würde auch gern rechnen können, ihn schreckt aber ihre Isolation ab. Weder die Funktionalität für das Tanzen noch für das Rechnen darf in diesem Hamster implementiert sein. Im Folgenden werden wir ihn „Adapterhamster“ nennen.

4. Schreibe ein geeignetes Hauptprogramm, das die Tanzhamster tanzen lässt, die Rechenhamster in geschlossenen Räumen einschließt und in regelmäßigen Abständen dem Benutzer die Gelegenheit gibt, alle Hamster zum Rechnen aufzufordern. Die Rechenhamster werden rechnen wie gewünscht, die Tanzhamster werden nur kundtun, dass sie nicht rechnen können. Der gesondert beschriebene einzelne Hamster tanzt mit den anderen Tanzhamstern wie einer von ihnen, kann aber trotzdem rechnen. Erzeuge ein geeignetes Territorium.

Bei einer Aufgabe von solcher Komplexität erscheint es sinnvoll, vor der Implementierung die Aufgabenstellung nach Hinweisen durchzusehen, wie die Klassenstruktur aussehen könnte. Dass es eine Klasse `TanzHamster` und eine Klasse `RechenHamster` geben muss, ist offensichtlich.
Wäre in Java Mehrfachvererbung erlaubt (wie z.B. in C++[13]), würden wir wahrscheinlich die Klasse `AdapterHamster` von den beiden anderen Hamsterklassen ableiten, um die Vorgaben zu erfüllen. Dies können wir in Java nicht, also müssen wir uns etwas anderes einfallen lassen. Da wir nur von einer Klasse ableiten dürfen, wählen wir erstmal `TanzHamster`, da der Adapterhamster „zu den Tanzhamstern gehören“ will, von den Rechenhamstern aber gerne nur das Können hätte. Letzteres Problem verschieben wir auf die Implementierung.
Dem erfahrenen Programmierer fällt weiterhin auf, dass alle vorkommenden Hamster zumindest ansprechbar in Bezug auf das Rechnen sein müssen. Konkret heißt das, sie müssen eine Methode rechnen() oder ähnliches zur Verfügung stellen. Dies ist eine typische Situation für eine besondere Möglichkeit, die wir in Java haben: ein Interface.
Erste Aufgabe ist also, ein Interface `AufsRechnenAnsprechbar` zu schreiben, welches alle von uns in diesem Beispiel benutzten Hamsterklassen implementieren müssen. Wir geben der Methode `rechne` zwei Parameter vom Typ `int` mit und lassen uns einen `int`-Wert zurückgeben, in dem die Reaktion des Hamsters ausgedrückt werden soll: Ein Hamster der Rechnen kann, wird das Ergebnis zurückgeben, ein Hamster der nicht rechnen kann, wird eine entsprechende *Exception* werfen.[14]

[13] vgl. z.B. [Bre05], S.303ff
[14] Zur Wiederholung empfehle ich [BB04], S.381 (Kapitel 13 Fehlerbehandlung mit Exceptions)

```
package entwurfsmuster.adapter;

public interface AufsRechnenAnsprechbar {
    public int rechne(int param1, int param2) throws RechnenException;
}
```

Die Klasse `RechnenException` wird von `RunTimeException` abgeleitet:

```
package entwurfsmuster.adapter;

public class RechnenException extends RuntimeException {
    public RechnenException(String message) {
        super(message);
    }
}
```

3.3.2 Das Territorium

Das Territorium sollte – zum Beispiel durch Mauern – die „Einsamkeit“der Rechenhamster darstellen und weiterhin den Tanzhamstern Platz zum Tanzen lassen. In Abbildung 3 ist das mit dieser Arbeit mitgelieferte Territorium zu sehen.

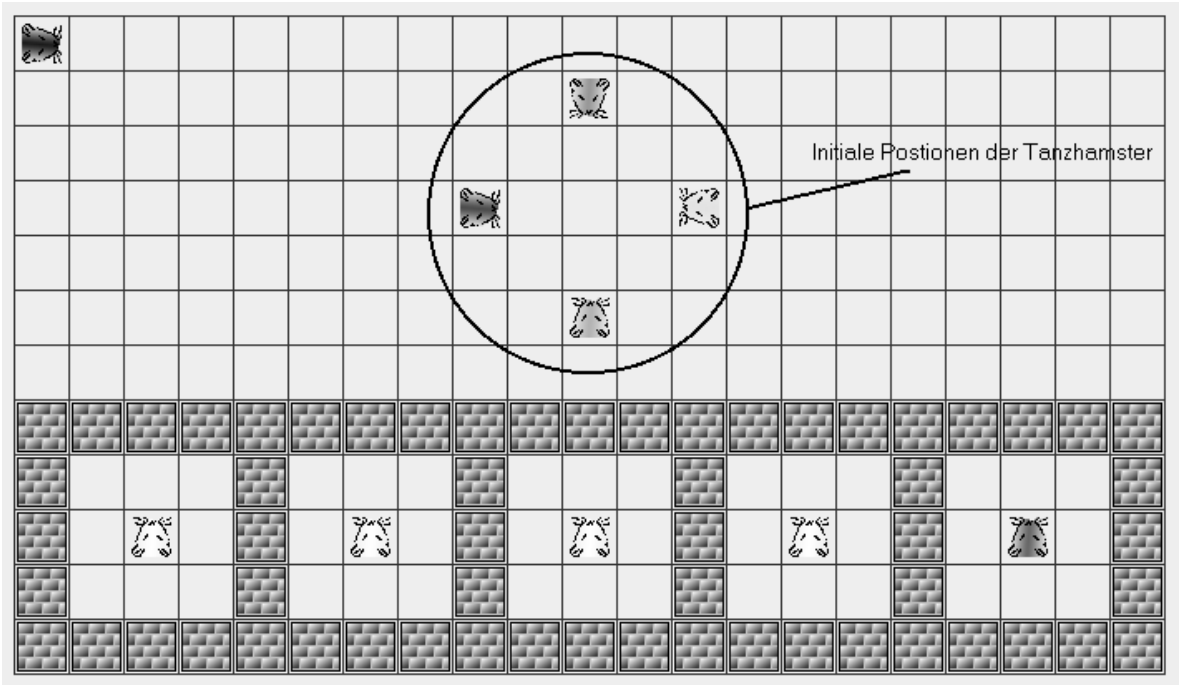


Abbildung 3: Territorium für das Hamsterprogramm zum Adapter-Muster

3.3.3 Die Tanzhamster

Wegen der Kürze werden wir für die Realisierung des Tanzens der Hamster eine simple Lösung wählen. Das soll den kreativen Leser jedoch nicht davon abhalten, sich einen komplexeren, „schöneren“ Algorithmus auszudenken.

Für den Hamstertanz verwenden wir vier Tanzhamster, deren initiale Positionen zueinander aus der Abbildung 3 zu entnehmen sind. Ein Tanzhamster geht nun während eines Tanzschritts einen Schritt vor, dreht sich einmal um sich selbst, geht einen Schritt zurück und wählt anschließend den nächsten Tanzhamster aus. Wir lassen das Hauptprogramm koordinieren, in welcher Reihenfolge die Tanzhamster ihren Tanzschritt ausführen dürfen.

```
package entwurfsmuster.adapter;

import entwurfsmuster.AllroundHamster;

public class TanzHamster extends AllroundHamster
                         implements AufsRechnenAnsprechbar {

    public TanzHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }

    public int rechne(int param1, int param2) throws RechnenException {
        throw new RechnenException("Ich kann nicht rechnen!");
    }

    public void tanze() {
        vor();                    //einen Schritt vor
        kehrt();                  //einmal um sich selbst drehen
        kehrt();
        kehrt();                 //ein Schritt rückwärts
        vor();
        kehrt();
    }
}
```

3.3.4 Die Rechenhamster

Dies ist die minimal notwendige Implementierung der `Klasse RechenHamster`:

```
package entwurfsmuster.adapter;

public class RechenHamster extends Hamster
                           implements AufsRechnenAnsprechbar {

    public RechenHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }

    public int rechne(int param1, int param2) {
        return param1 * param2;
    }
}
```

3.3.5 Der Adapterhamster

Nun kommen wir zur Lösung des Problems, das schon in der Aufgabenstellung angesprochen wurde. Wie kann der Adapterhamster rechnen, ohne dass er selbst die Funktion implementiert und ohne von `RechenHamster` abzuleiten?
Wir geben ihm ein Instanzattribut `rechenHamster` mit, das wir im Konstruktor initialisieren. Und in der Methode `rechne` geben wir die Aufgabe an das interne `RechenHamster`-Objekt weiter, wir *delegieren* also die Aufgabe. Das Übernehmen der Schnittstelle von der einen Klasse durch Vererbung und das Übernehmen der Funktionalität der anderen auf diesem Wege nennt man zusammen das **Adaptermuster**.

```
package entwurfsmuster.adapter;

public class AdapterHamster extends TanzHamster
                            implements AufsRechnenAnsprechbar {

    RechenHamster rechenHamster;

    public AdapterHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
        this.rechenHamster = new RechenHamster(9, 18, 0, 0);
    }

    public int rechne(int param1, int param2) {
        return rechenHamster.rechne(param1, param2);
    }
}
```

Das Adaptermuster lässt sich leider an einer Stelle nicht vollständig mit dem Hamstermodell abbilden: Der Begriff des „internen Objektes“ ist nur anhand der Klassenimplementierung zu erläutern, weil die Hamsterinstanzen grafisch in einer Weise dargestellt werden, die eine solche Strukturangabe ignoriert: Der interne Rechenhamster des Adapterhamsters taucht im Territorium genauso auf, wie der Adapterhamster selbst.

3.3.6 Das Hauptprogramm

Im Hauptprogramm werden zunächst alle Hamster erzeugt. In einer Endlosschleife geschieht nun immer nacheinander Folgendes:

1. Die Hilfsmethode `waehleTaenzer` wählt zufällig einen `TanzHamster` aus und dieser führt einen Tanzschritt aus. Wenn aber mehr als vier `TanzHamster`-Objekte im Territorium sind, wirft `waehleTaenzer` eine `ArrayIndexOutOfBoundsException`.[15] Diese wird abgefangen und durch eine neue `ZuvieleHamsterException` ersetzt, dessen Meldung den Grund für den Ausnahmefehler angibt.
   Dies geschieht mittels einer `for`-Schleife zehn Mal.

2. Der StandardHamster wird zur Kommunikation mit dem Benutzer eingesetzt. Er fragt ab, welcher Hamster welche Faktoren multiplizieren soll.

[15] mehr zu dieser Exception siehe [BB04], S.404 (Kapitel 13.6 Unchecked-Exceptions)

3. Eine `switch`-Anweisung bestimmt mit Hilfe der Eingabe das richtige Objekt (hier wird wieder der Vorteil genutzt, dass alle Hamster das Interface `AufsRechnenAnsprechbar` implementieren).

4. Die Methode `rechne` des Kandidaten wird ausgeführt. Die eventuelle Exception von nicht rechnen könnenden Hamstern wird abgefangen, sie geben eine entsprechende Meldung aus. Falls ein Ergebnis errechnet werden konnte, wird dies vom Kandidaten ausgegeben. Der Compiler kann nicht wissen, dass alle Klassen, die `AufsRechnenAnsprechbar` implementieren, über eine Methode `schreib` verfügen (weil sie Unterklassen von der `Hamster`-Klasse sind). Aus diesem Grund müssen wir vor dem Aufruf der Methode eine explizite Typumwandlung durchführen.

```
import entwurfsmuster.adapter.TanzHamster;
import entwurfsmuster.adapter.RechenHamster;
import entwurfsmuster.adapter.AdapterHamster;
import entwurfsmuster.adapter.AufsRechnenAnsprechbar;
import entwurfsmuster.adapter.RechnenException;
import entwurfsmuster.adapter.ZuvieleHamsterException;

public TanzHamster waehleTaenzer() throws ArrayIndexOutOfBoundsException {
    int zaehler = 0;
    Hamster[] alleHamster = Territorium.getHamster();
    TanzHamster[] andereTanzHamster = new TanzHamster[4];
    for (Hamster hamster : alleHamster) {
        if (hamster instanceof TanzHamster) {
            andereTanzHamster[zaehler] = (TanzHamster) hamster;
            zaehler++;
        }
    }
    int zufallsZahl = (int) (Math.random() * 4);
    return andereTanzHamster[zufallsZahl];
}

void main() {
    TanzHamster taenzer1 = new TanzHamster(3, 8, Hamster.OST, 0);
    TanzHamster taenzer2 = new TanzHamster(1, 10, Hamster.SUED, 0);
    TanzHamster taenzer3 = new TanzHamster(3, 12, Hamster.WEST, 0);
    TanzHamster taenzer4 = new AdapterHamster(5, 10, Hamster.NORD, 0);
    RechenHamster rechner1 = new RechenHamster(9, 2, Hamster.NORD, 0);
    RechenHamster rechner2 = new RechenHamster(9, 6, Hamster.NORD, 0);
    RechenHamster rechner3 = new RechenHamster(9, 10, Hamster.NORD, 0);
    RechenHamster rechner4 = new RechenHamster(9, 14, Hamster.NORD, 0);
    while (true) {
        for (int i = 0; i < 10; i++) {
            try {
                waehleTaenzer().tanze();
            } catch (ArrayIndexOutOfBoundsException e) {
                throw new ZuvieleHamsterException("Zuviele Tanzhamster "
                                                  + "im Territorium!");
```

```
            }
        }
        Hamster moderator = Hamster.getStandardHamster();
        int hamsterZahl =
                moderator.liesZahl("Welcher Hamster soll rechnen?");
        int faktor1 = moderator.liesZahl("Bitte 1. Faktor eingeben.");
        int faktor2 = moderator.liesZahl("Bitte 2. Faktor eingeben.");
        AufsRechnenAnsprechbar kandidat;
        switch (hamsterZahl) {
        case 0:
            kandidat = taenzer1;
            break;
        case 1:
            kandidat = taenzer2;
            break;
        case 2:
            kandidat = taenzer3;
            break;
        case 3:
            kandidat = taenzer4;
            break;
        case 4:
            kandidat = rechner1;
            break;
        case 5:
            kandidat = rechner2;
            break;
        case 6:
            kandidat = rechner3;
            break;
        case 7:
            kandidat = rechner4;
            break;
        default:
            kandidat = rechner1;
        }
        try {
            int ergebnis = kandidat.rechne(faktor1, faktor2);
            ((Hamster)kandidat).schreib("Das Ergebnis lautet " + ergebnis);
        } catch (RechnenException e) {
            ((Hamster)kandidat).schreib(e.getMessage());
        }
    }
}
```

Die `ZuvieleHamsterException` ist kongruent zur `RechnenException`:

```
package entwurfsmuster.adapter;
```

```
public class ZuvieleHamsterException extends RuntimeException {
    public ZuvieleHamsterException(String message) {
        super (message);
    }
}
```

3.3.7 Zusammenfassung und Ausblick

Das Entwurfsmuster Adapter ist ein objektbasiertes Strukturmuster.[16] Es wird hauptsächlich dazu verwendet, die Schnittstelle einer Klasse an eine andere – erwartete – anzupassen, um die Zusammenarbeit zwischen Klassen zu ermöglichen, die sonst aufgrund nicht kompatibler Schnittstellen nicht dazu in der Lage wären. Abbildung 4 zeigt ein Klassendiagramm, das die wesentlichen Merkmale des Musters abstrakt darstellt.

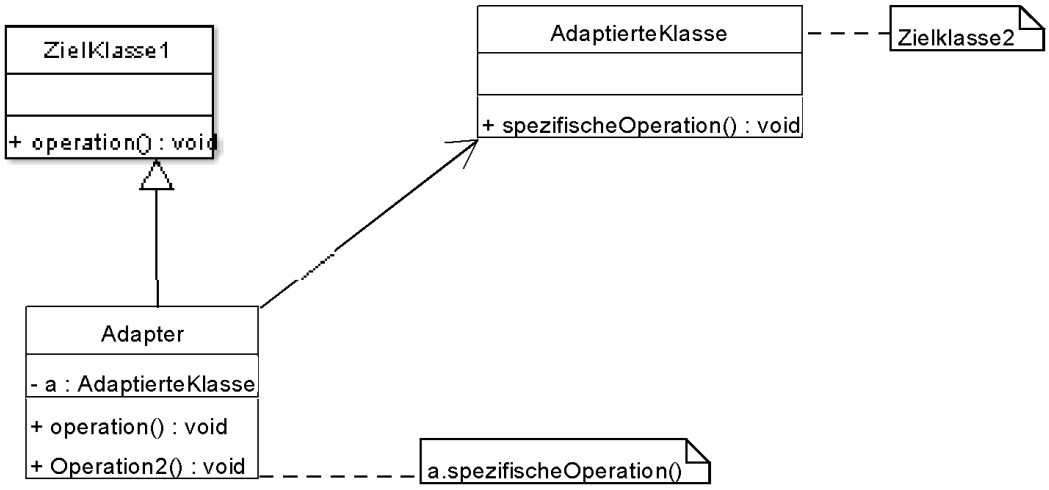


Abbildung 4: Klassendiagramm zum Entwurfsmuster Adapter

In dem hier vorgestellten Hamsterprogramm erscheint es künstlich, dem Adapterhamster die Multiplikationsfunktion zu verbieten, um das Adapter-Muster einführen zu können und für eine Zeile Code scheint es übertrieben, auf Wiederverwendbarkeit zu pochen.
Doch im Programmieralltag können uns viel komplexere Methoden begegnen, so dass das Adapter-Muster uns nicht nur eine, sondern tausende Zeilen an Code sparen kann. Außerdem kann es vorkommen, dass wir nur die kompilierte Version einer Klasse zur Verfügung haben und selbst wenn wir wollten, keinen Quellcode aus ihr herauskopieren könnten. Dieses Problem ist namensgebend für das Adaptermuster – mit seiner Hilfe können wir einen Adapter zu jener Klasse schaffen und so ihre Funktionalität nutzen.

[16] Bei Sprachen die Mehrfachvererbung erlauben, kann ein Adapter auch klassenbasiert sein. Siehe dazu [GHJV04], S.171ff)

3.4 Abstrakte Fabrik (Abstract Factory)

3.4.1 Motivationsaufgabe

Als Motivation für die Einführung des nächsten Entwurfsmusters schauen wir uns zunächst die Nachteile der Implementierung der letzten Aufgabe an:

1. Es werden relativ viele Objekte im Hauptprogramm erzeugt. In unserem kleinen Beispiel mag es noch annehmbar sein, aber bei größeren Programmen mit 1000 statt 10 Objekten kann dies zu fehlerbegünstigender Unübersichtlichkeit führen. Wünschenswert wäre eine Hamstererzeugung außerhalb des Hauptprogramms.

2. Es fällt auf, dass die `switch`-Anweisung durch eine einzige Zeile zu ersetzen wäre, wenn alle Objekte, um die es geht, in einem Array untergebracht wären: Dann könne man über den Index direkt auf das Wunschobjekt zugreifen.

3. Wenn wir einzelne Hamsterklassen ändern, verbessern, erweitern (und dabei die alten Klassen behalten wollen, also nicht überschreibem), müssen wir umständlich alle `new`-Anweisungen im Hauptprogramm ändern. Wieder ist es zum Verständnis hilfreich, sich ein großes, komplexes Programm vorzustellen, wo womöglich an den verschiedensten Stellen new-Anweisungen auftauchen, die wir in so einem Fall alle einzeln suchen, finden und anpassen müssen.

Hauptziel ist also, aufgrund der Übersichtlichkeit (und auch der Wiederverwendbarkeit, wie wir später sehen werden), die Objekterzeugung vom sonstigen Programmablauf zu trennen. Eine Möglichkeit, dies zu tun, ist eine **Abstrakte Fabrik**; dieses Erzeugermuster möchte ich nun schrittweise einführen.

3.4.2 Territorium

Es wird dasselbe Territorium verwendet wie im vorherigen Kapitel.

3.4.3 Lösung A: Auslagerung der Objekterzeugung

Als erstes implementieren wir eine Methode `erzeugeVieleHamster` außerhalb des Hauptprogramms, in die alle `new`-Anweisungen verschoben werden.
Diese muss nun die erzeugten Objekte in einer geeigneten Form an das die Methode ausführende Hauptprogramm zurückgeben. Was in diesem Fall „geeignet“ heißt, hängt davon ab, um was für Objekte es sich handelt und wie sie verwendet werden sollen. Wir erinnern uns an unseren zweiten Kritikpunkt: Es sind alles `Hamster`-Objekte, die am besten in einem Array verfügbar sind.
Diese Methode wird nun zu Beginn des Hauptprogramms ausgeführt, die `switch`-Anweisung wird – wie in der Kritik vorgeschlagen – durch einen direkten Zugriff auf das Array ersetzt:

```
import java.util.List;
import java.util.ArrayList;
import entwurfsmuster.adapter.TanzHamster;
import entwurfsmuster.adapter.RechenHamster;
import entwurfsmuster.adapter.AdapterHamster;
import entwurfsmuster.adapter.AufsRechnenAnsprechbar;
import entwurfsmuster.adapter.RechnenException;
```

```
import entwurfsmuster.adapter.ZuvieleHamsterException;

public AufsRechnenAnsprechbar[] erzeugeVieleHamster() {
    List<AufsRechnenAnsprechbar> alleHamster =
            new ArrayList<AufsRechnenAnsprechbar>();
    alleHamster.add(new TanzHamster(3, 8, Hamster.OST, 0));
    alleHamster.add(new TanzHamster(1, 10, Hamster.SUED, 0));
    alleHamster.add(new TanzHamster(3, 12, Hamster.WEST, 0));
    alleHamster.add(new AdapterHamster(5, 10, Hamster.NORD, 0));
    alleHamster.add(new RechenHamster(9, 2, Hamster.NORD, 0));
    alleHamster.add(new RechenHamster(9, 6, Hamster.NORD, 0));
    alleHamster.add(new RechenHamster(9, 10, Hamster.NORD, 0));
    alleHamster.add(new RechenHamster(9, 14, Hamster.NORD, 0));
    return alleHamster.toArray(new AufsRechnenAnsprechbar[0]);
}

public TanzHamster waehleTaenzer() throws ArrayIndexOutOfBoundsException {
    int zaehler = 0;
    Hamster[] alleHamster = Territorium.getHamster();
    TanzHamster[] andereTanzHamster = new TanzHamster[4];
    for (Hamster hamster : alleHamster) {
        if (hamster instanceof TanzHamster) {
            andereTanzHamster[zaehler] = (TanzHamster) hamster;
            zaehler++;
        }
    }
    int zufallsZahl = (int) (Math.random() * 4);
    return andereTanzHamster[zufallsZahl];
}

void main() {
    AufsRechnenAnsprechbar[] hamsterArray = erzeugeVieleHamster();
    while (true) {
        for (int i = 0; i < 10; i++) {
            try {
                waehleTaenzer().tanze();
            } catch (ArrayIndexOutOfBoundsException e) {
                throw new ZuvieleHamsterException ("Zuviele Tanzhamster "
                                                    + "im Territorium!");
            }
        }
        Hamster moderator = Hamster.getStandardHamster();
        int hamsterZahl =
                moderator.liesZahl("Welcher Hamster soll rechnen (1-8)?");
        int param1 = moderator.liesZahl ("Bitte 1. Zahl eingeben.");
        int param2 = moderator.liesZahl ("Bitte 2. Zahl eingeben.");
        AufsRechnenAnsprechbar kandidat = hamsterArray [hamsterZahl - 1];
        try {
```

```
            int ergebnis = kandidat.rechne (param1, param2);
            ((Hamster)kandidat).schreib ("Das Ergebnis lautet " + ergebnis);
        } catch (Exception e) {
            ((Hamster)kandidat).schreib (e.getMessage());
        }
    }
}
```

Noch eine Bemerkung zur Methode `erzeugeVieleHamster`: Zuerst eine ArrayList zu verwenden und erst zum Schluss in ein Array umzuwandeln, hat den Vorteil, dass man nicht vorher festlegen muss, wie viele Objekte das Array beinhaltet.
Die ersten beiden Kritikpunkte sind nun beseitigt – ich weise darauf hin, dass auch eine Menge zusätzlicher Hamster und dadurch eine Menge zusätzlicher Objekte die `main`-Methode nicht vergrößern würden. Doch dem dritten Kritikpunkt sind wir noch nicht konsequent genug entgegengetreten, da sich das dort beschriebene Problem von der `main`-Methode auf die Hilfsmethode verlagert hat.

3.4.4 Lösung B: Eine Hamsterfabrik

Unser nächster Schritt ist die Implementierung einer neuen Klasse `HamsterFabrik`, die unsere Hamstererzeugungen in Methoden, welche die Objekte zurückliefern, zur Verfügung stellt:

```
package entwurfsmuster.abstrakteFabrik.abstrakteFabrikB;

import entwurfsmuster.adapter.TanzHamster;
import entwurfsmuster.adapter.RechenHamster;
import entwurfsmuster.adapter.AdapterHamster;

public class HamsterFabrik {

    public TanzHamster erzeugeTanzHamster(int r, int s, int b, int k) {
        return new TanzHamster(r, s, b, k);
    }

    public RechenHamster erzeugeRechenHamster(int r, int s, int b, int k) {
        return new RechenHamster(r, s, b, k);
    }

    public AdapterHamster erzeugeAdapterHamster(int r, int s, int b, int k) {
        return new AdapterHamster(r, s, b, k);
    }
}
```

Dadurch sind nun folgende weitere Änderungen möglich:

- Es wird in der `main`-Methode als erstes eine Instanz der Klasse `HamsterFabrik` erzeugt.
- Diese wird der Methode `erzeugeVieleHamster` als Parameter mitgegeben.
- Statt der `new`-Aufrufe werden jetzt in `erzeugeVieleHamster` die Methoden der Fabrik benutzt, um die Objekte zu erzeugen.

```
import java.util.List;
import java.util.ArrayList;
import entwurfsmuster.adapter.TanzHamster;
import entwurfsmuster.adapter.RechenHamster;
import entwurfsmuster.adapter.AdapterHamster;
import entwurfsmuster.adapter.AufsRechnenAnsprechbar;
import entwurfsmuster.adapter.RechnenException;
import entwurfsmuster.adapter.ZuvieleHamsterException;
import entwurfsmuster.abstrakteFabrik.abstrakteFabrikB.HamsterFabrik;

public AufsRechnenAnsprechbar[] erzeugeVieleHamster(HamsterFabrik fabrik) {
    List<AufsRechnenAnsprechbar> alleHamster =
            new ArrayList<AufsRechnenAnsprechbar>();
    alleHamster.add(fabrik.erzeugeTanzHamster(3, 8, Hamster.OST, 0));
    alleHamster.add(fabrik.erzeugeTanzHamster(1, 10, Hamster.SUED, 0));
    alleHamster.add(fabrik.erzeugeTanzHamster(3, 12, Hamster.WEST, 0));
    alleHamster.add(fabrik.erzeugeAdapterHamster(5, 10, Hamster.NORD, 0));
    alleHamster.add(fabrik.erzeugeRechenHamster(9, 2, Hamster.NORD, 0));
    alleHamster.add(fabrik.erzeugeRechenHamster(9, 6, Hamster.NORD, 0));
    alleHamster.add(fabrik.erzeugeRechenHamster(9, 10, Hamster.NORD, 0));
    alleHamster.add(fabrik.erzeugeRechenHamster(9, 14, Hamster.NORD, 0));
    return alleHamster.toArray(new AufsRechnenAnsprechbar[0]);
}

public TanzHamster waehleTaenzer()
        throws ArrayIndexOutOfBoundsException {...}

void main() {
    HamsterFabrik fabrik = new HamsterFabrik();
    AufsRechnenAnsprechbar[] hamsterArray = erzeugeVieleHamster (fabrik);
    while (true) {...}
}
```

3.4.5 Lösung C: Erhöhte Wiederverwendbarkeit durch Unterklassenbildung

Was haben wir durch diese neue Klasse – die Fabrik – gewonnen?
Unsere Ausgangsfragestellung war: Wie können wir uns die Arbeit erleichtern und übersichtlicher machen, wenn wir nun neue, variierte Hamster implementieren und diese statt den alten im Hauptprogramm verwenden wollen?
Die Antwort auf diese Frage ist sozusagen die Pointe dieses Entwurfsmusters, dessen weitere Struktur in folgenden drei Schritten offenbar wird:

1. Alle veränderten Hamster werden als Unterklassen der alten Hamster implementiert, also zum Beispiel wird `AddierHamster` von `RechenHamster` abgeleitet.

2. Ebenso wird eine Klasse `AddierHamsterFabrik` als Unterklasse von `HamsterFabrik` implementiert, in der die alten Fabrikmethoden überschrieben werden. Dabei ist auf Folgendes zu achten:

(a) Wichtig ist, dass *wirklich* die alten Methoden überschrieben werden. Um beim Beispiel zu bleiben: Es soll keine Methode `erzeugeAddierHamster` geben, sondern `erzeugeRechenHamster` wird überschrieben.

(b) In der Methode wird eine Instanz von `AddierHamster` erzeugt...

(c) ...aber es wird eine Instanz von `RechenHamster` zurückgegeben. Das ergibt sich natürlich schon aus a), es erscheint mir aber – zum besseren Verständnis des Wirkens des Entwurfsmusters – wichtig, hier auf die Anwendung von Polymorphie hinzuweisen.

Eine der überschriebenen Fabrikmethoden in `AddierHamsterFabrik` sähe also so aus:

```
public RechenHamster erzeugeRechenHamster(int r, int s, int b, int k) {
  return new AddierHamster (r, s, b, k);
}
```

3. Im Hauptprogramm muss *nur eine* Zeile geändert werden, um `alle` neuen Hamster zu verwenden:
   Die Zeile

   ```
   HamsterFabrik hamsterFabrik = new HamsterFabrik();
   ```

   wird durch

   ```
   HamsterFabrik hamsterFabrik = new AddierHamsterFabrik();
   ```

   ersetzt.

Übrigens müssen in einem Durchgang nicht alle Hamster ersetzt werden. In der neuen Hamsterfabrik werden dann nur die Fabrikmethoden überschrieben, für die es neue Hamsterklassen gibt.
Ich werde im Folgenden demonstrieren, wie nun der Rechenhamster durch einen anderen ersetzt wird, der addiert, statt multipliziert.
Die Klasse `AddierHamster` überschreibt die Methode `rechnen`:

```
package entwurfsmuster.abstrakteFabrik.abstrakteFabrikC;

import entwurfsmuster.adapter.RechenHamster;

public class AddierHamster extends RechenHamster {

    public AddierHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }

    public int rechnen(int param1, int param2) {
        return param1 + param2;
    }
}
```

Die neue Fabrik `AddierHamsterFabrik` überschreibt nur `erzeugeRechenHamster` und übernimmt alle anderen geerbten Methoden unverändert:

```
package entwurfsmuster.abstrakteFabrik.abstrakteFabrikC;

import entwurfsmuster.adapter.RechenHamster;
import entwurfsmuster.abstrakteFabrik.abstrakteFabrikB.HamsterFabrik;

public class AddierHamsterFabrik extends HamsterFabrik {

    public RechenHamster erzeugeRechenHamster(int r, int s, int b, int k) {
        return new AddierHamster(r, s, b, k);
    }
}
```

Das Hauptprogramm wird an dieser Stelle nicht nochmal dargestellt, da hier nur die oben aufgeführte Zeile geändert werden muss.

3.4.6 Zusammenfassung und Ausblick

Das Entwurfsmuster Abstrakte Fabrik ist ein objektbasiertes Erzeugungsmuster. Abbildung 5 zeigt ein Klassendiagramm, das abstrakt die wesentlichen Merkmale des Musters zusammenfasst.

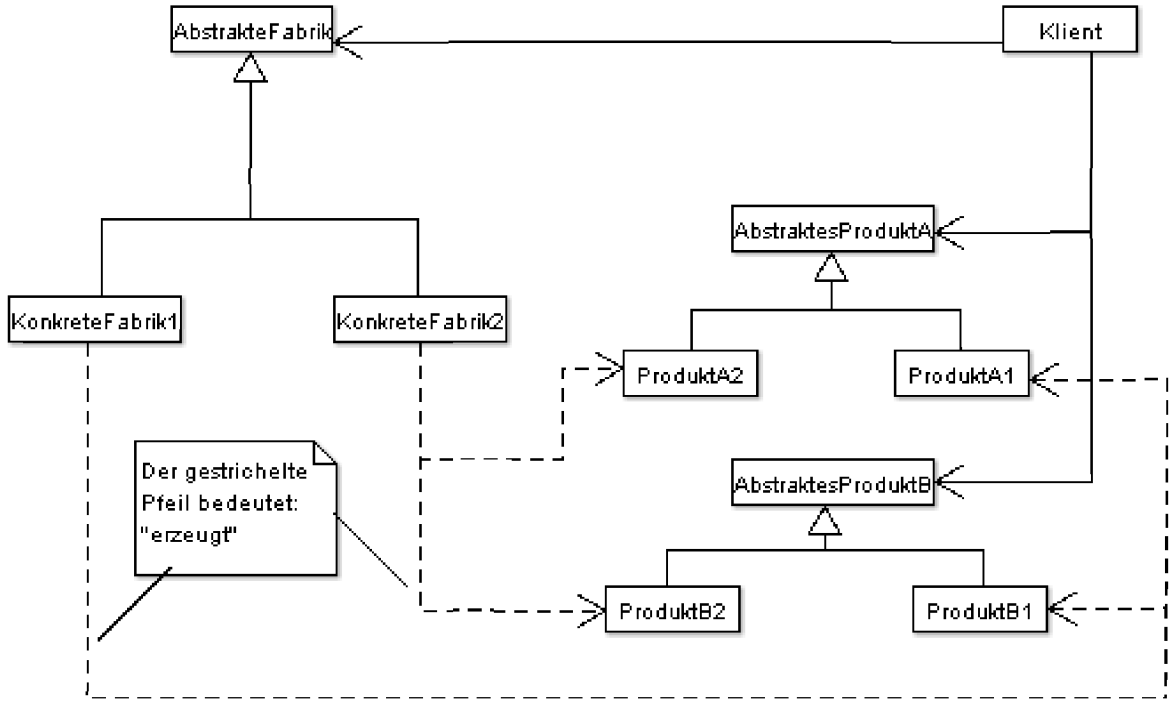


Abbildung 5: Klassendiagramm zum Entwurfsmuster Abstrakte Fabrik

Durch Anwendung des Entwurfsmusters ist zunächst ein größerer Programmieraufwand gegeben, aber auch der Vorteil der größeren Übersichtlichkeit, da die Fabriken sich auf die Erzeugung konzentrieren und das Hauptprogramm sich nur entscheiden muss, welche Fabrik die Hamster erzeugen soll, und damit, welche Hamster erzeugt werden.
Zudem erleichtert der Einsatz von Fabriken die Arbeitsteilung, da sich jetzt ein Programmierer z.B. nur um die Erzeugung und Bereitstellung von Objekten kümmern kann, während ein anderer ihre Verwendung implementiert.

Außerdem ist zu bemerken, dass in diesem Kapitel eigentlich *zwei* Entwurfsmuster vorgestellt wurden. Das Prinzip der Abstrakten Fabrik besteht im Grunde darin, die Objekterzeugung in speziell dafür angelegte Klassen auszulagern und gezielt die Vorteile auszunutzen, die sich aus ihrer Vererbungshierarchie ergeben können.
Wie aber die Objekterzeugung in diesen Fabriken abläuft, ist nicht in diesem Muster festgelegt. Der Einsatz von den hier vorgestellten `erzeugeX`-Methoden ist ein extra Entwurfsmuster und heißt **Fabrikmethode**. Es wäre allerdings auch möglich statt dessen das Entwurfsmuster **Prototyp** einzusetzen – letzteres wird in dieser Arbeit nicht erläutert, es ist nachzulesen in [GHJV04], S.144ff.
Schließlich soll an dieser Stelle darauf hingewiesen werden, dass eine Anwendung meistens nur ein Exemplar einer konkreten Fabrik benötigt. Deswegen sind diese oft Singletons.

3.5 Zustand (State)

3.5.1 Motivationsaufgabe

Es soll ein Apportierhamster implementiert werden, der auf Verhalten ihm gegenüber mit eigenem Verhalten reagiert und das ungefähr so, wie es ein Hund tun würde. Das bedeutet konkret, dass der Apportierhamster über Folgendes verfügen soll:

1. Er soll Dinge tun können, die Hunde üblicherweise tun: fressen, apportieren, usw.
2. Es soll Möglichkeiten geben, mit dem Apportierhamster in Kontakt zu treten, d.h. man soll ihn füttern und streicheln können usw.
3. In geeigneter Weise soll spezifiziert werden, wie sich der Apportierhamster gerade fühlt und was er gerade tut. Davon soll abhängig gemacht werden, wie der Apportierhamster auf Verhalten ihm gegenüber reagiert. Beispielsweise wird der Apportierhamster sehr wahrscheinlich fressen, wenn er gefüttert wird, aber nicht, wenn er gerade schläft.

Außerdem soll der ApportierHamster leicht erweiterbar sein: Das Verhalten eines Hundes kann sehr komplex werden, wenn man versucht, es vollständig abzubilden.

3.5.2 Zum Territorium

Das Territorium für dieses Beispiel kann beliebig frei gestaltet sein – einzige Bedingung: es ist darauf zu achten, dass wirklich ein Korn auf der Kachel liegt, das man dem Apportierhamster zum Fressen oder Apportieren zuweist.

3.5.3 Schritt A: intuitive Vorgehensweise

Ich definiere in der Klasse `ApportierHamster` die öffentlichen Methoden `wecke`, `streichle`, `fuettere` und `lasseApportieren`, auf die man von außen zugreifen kann, um in Kontakt mit dem Apportierhamster zu treten. Weiterhin deklariere ich drei private Methoden `fresse`, `apportiere` und `wacheAuf`, in denen ich das Verhalten des Hamsters definiere. In den drei Instanzattributen vom Typ `boolean satt`, `schlaeft` und `zufrieden` halte ich das Befinden des Apportierhamsters fest.
Mit einer bedingten Anweisung in jeder der öffentlichen Methoden kann ich nun definieren, wie sich der Hamster verhalten und wie sich sein Befinden ändern soll. Im Konstruktor lege ich sein Anfangsverhalten fest: Er schläft, ist zufrieden und satt.

```
package entwurfsmuster.zustand.zustandA;

import entwurfsmuster.AllroundHamster;

public class ApportierHamster extends AllroundHamster {

    private boolean satt;
    private boolean schlaeft;
    private boolean zufrieden;

    public ApportierHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
```

```
        this.satt = true;
        this.schlaeft = true;
        this.zufrieden = true;
    }

    public void streichle() {
        if (!this.schlaeft) {
            schreib("***Schwanz wedeln***");
            if (this.satt) {
                this.zufrieden = true;
            } else {
                schreib("***keine Reaktion***");
            }
        }
}

    public void wecke() {
        if (this.schlaeft) {
            this.wacheAuf();
            this.schlaeft = false;
        } else {
            schreib("***keine Reaktion***");
        }
    }

    public void fuettere(int reihe, int spalte) {
        if (!this.schlaeft) {
            fresse(reihe, spalte);
            this.satt = true;
        } else {
            schreib("***keine Reaktion***");
        }
    }

    public void lasseApportieren(int reihe, int spalte) {
        if (!this.schlaeft && this.zufrieden && this.satt) {
            apportiere(reihe, spalte);
            this.satt = false;
        } else if (!this.schlaeft && this.zufrieden && !this.satt) {
            apportiere(reihe, spalte);
            this.zufrieden = false;
        } else if (!this.schlaeft && !this.zufrieden && !this.satt) {
            fresse(reihe, spalte);
        } else { // this.schlaeft = true
            schreib("***keine Reaktion***");
        }
    }
```

```
    private void fresse(int reihe, int spalte) {
        int startReihe = getReihe();
        int startSpalte = getSpalte();
        gotoKachel(reihe, spalte);
        if (kornDa()) {
            nimm();
        }
        gotoKachel(startReihe, startSpalte);
    }

    private void apportiere(int reihe, int spalte) {
        fresse(reihe, spalte);
        if (!maulLeer()) {
            gib();
        }
    }

    private void wacheAuf() {
        kehrt();
        kehrt();
    }
}
```

Ein einfaches Hauptprogramm zum Test wäre zum Beispiel:

```
import entwurfsmuster.zustand.zustandA.ApportierHamster;

void main() {
    ApportierHamster waldi = new ApportierHamster(2, 2, Hamster.SUED, 0);
    waldi.wecke();
    waldi.lasseApportieren(5, 5);
    waldi.lasseApportieren(7, 7);
    waldi.fuettere(2,3);
    waldi.streichle();
}
```

Beim Betrachten der Methode `lasseApportieren` fällt auf, was passieren wird, wenn die möglichen Zustände und Aktivitäten erweitert werden. Die Bedingungsanweisungen werden immer komplexer werden, da es immer mehr Kombinationen gibt und damit wird die Klasse groß und unübersichtlich.
Wir müssen nun überlegen, welchen Teil der Implementierung wir auslagern können bzw. wie wir die Implementierung umstrukturieren können, um dem Apportierhamster auch komplexeres Verhalten mitgeben zu können.

3.5.4 Schritt B: Kapselung des Zustands

Vorbemerkung: Die Version der Klasse `ApportierHamster` dieses Kapitels ist nicht kompilierbar. Dies ließ sich nicht vermeiden, da hier nur ein Zwischenschritt erläutert wird, der aber für das Verständnis dieses Entwurfsmusters unabdingbar ist.

Eine Möglichkeit ist, den Zustand des Hamsters nicht in boole'schen Attributen auszudrücken, sondern dafür eine eigene Klasse `HamsterZustand` zu schaffen, und dem `ApportierHamster` ein Instanzattribut `zustand` vom Typ dieser Klasse zu geben. Die Klasse `HamsterZustand` muss alle öffentlichen Methoden implementieren, die die Klasse `ApportierHamster` auch hat. In `ApportierHamster` muss nun z.B. bei der Methode `fuettere` nur noch die Methode `fuettere` von `HamsterZustand` aufgerufen werden und dieser sorgt dann dafür, dass auch das dem Zustand entsprechende Verhalten eintritt.
Bis hier hin sieht die gewählte Vorgehensweise dem Adapter-Muster sehr ähnlich. Jetzt kommt jedoch etwas Neues hinzu: Es gibt nicht nur einen Zustand, in dem sich der Hamster befinden kann, sondern mehrere (auf unsere vorherigen booleschen Attribute bezogen maximal soviele, wie man Kombinationen von ihren Belegungen bilden kann). Das bedeutet, dass wir von einer Klasse `HamsterZustand` so viele Unterklassen ableiten müssen, wie es Zustände gibt und sich hinter dem neuen Attribut `zustand` vom Typ `HamsterZustand` mittels Polymorphie eine konkrete Unterklasse versteckt.
Die gerade beschriebene Vorgehensweise nennt man **Zustands-Muster**; wir werden sie nun Schritt für Schritt durchgehen:
Schauen wir uns nun zunächst die neue `ApportierHamster`-Klasse an, ohne uns um die genaue Implementierug der Zustandsklassen zu kümmern (deswegen verwenden wir für die Initialisierung des `zustand`-Attributs einen Platzhalter):

```
package entwurfsmuster.zustand.zustandB;

import entwurfsmuster.AllroundHamster;

public class ApportierHamster extends AllroundHamster {

    HamsterZustand zustand;

    public ApportierHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
        this.zustand = konkreterZustand; //Platzhalter, so nicht kompilierbar
    }

    public void streichle() {
        zustand.streichle();
    }

    public void wecke() {
        zustand.wecke();
    }

    public void fuettere(int reihe, int spalte) {
        zustand.fuettere(reihe, spalte);
    }

    public void lasseApportieren(int reihe, int spalte) {
        zustand.lasseApportieren(reihe, spalte);
    }
```

```
    private void fresse(int reihe, int spalte) {...}
    private void apportiere(int reihe, int spalte) {...}
    private void wacheAuf() {...}
}
```

Die Klasse ist kleiner geworden und sie wird – relativ zum ersten Lösungsversuch – auch bei Erweiterung klein bleiben: Für alle Aktivitäten des Apportierhamsters müssen zwar alle Methoden hier nach wie vor implementiert werden, aber alle anderen Methoden sind Einzeiler. Alle Informationen über den Zustand sind ebenfalls gekapselt, d.h. sie können beliebig groß werden, ohne dass dadurch die Komplexität der Klasse `ApportierHamster` größer wird.
Wenden wir uns nun der neuen Klasse `HamsterZustand` zu. Wie schon erwähnt, soll sie nur eine abstrakte Oberklasse für konkrete Hamsterzustände sein:

```
package entwurfsmuster.zustand.zustandB;

abstract public class HamsterZustand extends Hamster {
    abstract public void streichle();
    abstract public void wecke();
    abstract public void lasseApportieren(int reihe, int spalte);
    abstract public void fuettere(int reihe, int spalte);
}
```

Man könnte sich jetzt fragen, warum dies kein Interface ist. Es ist deswegen keines, weil es in einer späteren Version um Implementierungen erweitert wird, so dass es eine abstrakte Klasse sein muss.
Als nächstes wollen wir den ersten konkreten Zustand schreiben. Für die Benennung der Zustände bleiben wir der Einfachheit halber bei den Begriffen „schlaeft“, „satt“ und „zufrieden“ und setzen den Klassennamen eines jeweiligen Zustands aus den Worten zusammen, die auf ihn zu treffen; der Initialzustand des Apportierhamsters ist also `SchlaeftSattZufrieden`.
Setzen wir nun diese Klasse Schritt für Schritt zusammen und beginnen mit der Frage, ob ein Konstruktor benötigt wird. Das ist nicht der Fall, denn es müssen keine Attribute initialisiert werden, es geht nur um das variierte Überschreiben von Methoden. Es genügt also eigentlich, den Superkonstruktor implizit aufrufen zu lassen.
Jedoch muss noch ein anderer Sachverhalt betrachtet werden: Nehmen wir an, der Apportierhamster befindet sich in Zustand X, wechselt in Zustand Y und danach wieder nach X. Dann würde bei dieser Vorgehensweise immer ein neues Objekt instanziiert werden, obwohl das eigentlich immer nur dann nötig ist, wenn ein Zustand zum ersten Mal verwendet wird. Mit anderen Worten: Wir benötigen immer nur eine Instanz einer konkreten Zustandsklasse und verwenden daher das Entwurfsmuster *Singleton* (siehe Kapitel 3.2, Seite 37). Unsere Klasse `SchlaeftSattZufrieden` sieht bisher so aus:

```
package entwurfsmuster.zustand.zustandB;

public class SchlaeftSattZufrieden extends HamsterZustand {

    private static SchlaeftSattZufrieden exemplar;

    public static HamsterZustand exemplar() {
```

```
        if (exemplar == null) {
            exemplar = new SchlaeftSattZufrieden();
        }
        return exemplar;
    }
}
```

Noch eine Bemerkung zum Typ des Rückgabewerts der `exemplar`-Methode: Wir beziehen uns hier auf die Oberklasse, weil sie vom `ApportierHamster` erwartet wird.
Wenn wir im Folgenden an das Überschreiben der Methoden gehen, taucht ein neues Problem auf: Wollen wir zum Beispiel unsere alte Methode `streichle` (im `ApportierHamster`) übersetzen, müssen wir zum einen den Hamster etwas schreiben lassen, zum anderen den Zustand wechseln:

```
    public void streicheln() {
        if (this.satt) {
            zufrieden = true;
        }
        schreib("***Schwanz wedeln***");
    }
```

Den Hamster etwas schreiben lassen können wir nur, wenn die Methode den Hamster kennt – das tut sie bisher aber nicht. Ebenfalls haben wir bisher auch noch keine Möglichkeit, den Zustand zu wechseln.

3.5.5 Schritt C: Doppelte Delegierung

Wir lösen diese Probleme in folgender Weise:

- Alle öffentlichen Methoden erhalten als Parameter das betreffende `ApportierHamster`-Objekt. Das müssen wir an drei Stellen berücksichtigen: Im abstrakten `HamsterZustand`, in seinen konkreten Unterklassen und im `ApportierHamster`, in der diese Methoden aufgerufen werden.

- Die Klasse `ApportierHamster` erhält eine neue Methode `aendereZustand`. Diese soll sie selbst aber gar nicht aufrufen, denn sie weiß ja nicht, wann ein Zustandswechsel stattfinden muss. Es handelt sich viel mehr um eine Art set-Methode für das `zustand`-Attribut der Klasse, damit dieses von außen geändert werden kann.

- In `HamsterZustand` wird dazu auch eine Methode `aendereZustand` implementiert, die im Gegensatz zu den anderen dort nicht abstrakt ist:

  ```
  public static void aendereZustand (ApportierHamster hamster,
                                     HamsterZustand zustand) {
      hamster.aendereZustand (zustand);
  }
  ```

  Damit haben alle konkreten Hamsterzustände die Möglichkeit, eine Zustandsänderung einzuleiten (denn sie haben das Wissen darüber, *wann* das geschehen soll), führen diese aber nicht selbst durch, sondern delegieren sie zurück an den Apportierhamster.

Schauen wir uns unsere drei geänderten Klassen an – zunächst den neuen `ApportierHamster`. Hier können wir jetzt auch den Platzhalter bei der Initialisierung des Attributs `zustand` durch unseren konkreten Hamsterzustand `SchlaeftSattZufrieden` ersetzen:

```
package entwurfsmuster.zustand.zustandC;

import entwurfsmuster.AllroundHamster;

public class ApportierHamster extends AllroundHamster {

    HamsterZustand zustand;

    public ApportierHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
        this.zustand = SchlaeftSattZufrieden.exemplar();
    }

    public void aendereZustand(HamsterZustand zustand) {
        this.zustand = zustand;
    }

    public void streichle() {
        zustand.streichle(this);
    }

    public void wecke() {
        zustand.wecke(this);
    }

    public void fuettere(int reihe, int spalte) {
        zustand.fuettere(this, reihe, spalte);
    }

    public void lasseApportieren(int reihe, int spalte) {
        zustand.lasseApportieren(this, reihe, spalte);
    }

    void fresse(int reihe, int spalte) {
        int startReihe = getReihe();
        int startSpalte = getSpalte();
        gotoKachel(reihe, spalte);
        if (kornDa()) {
            nimm();
        }
        gotoKachel(startReihe, startSpalte);
    }

    void apportiere(int reihe, int spalte) {
```

```
        fresse(reihe, spalte);
        if (!maulLeer()) {
            gib();
        }
    }

    void wacheAuf() {
        kehrt();
        kehrt();
    }
}
```

Bezüglich der Zugriffsrechte ist noch Folgendes zu bemerken: Die Methoden, die das Verhalten des Apportierhamsters selbst ausdrücken, dürfen nun nicht mehr `private` sein, damit der jeweilige Zustand, in dem sich der Hamster befindet, sie aufrufen kann.
Wenn alle Zustände und die Apportierhamsterklasse alle in einem Paket sind, genügt es, keinen Zugriffsrechtstyp anzugeben. Damit sind die Methoden innerhalb des Paketes öffentlich.[17]
Wenn sich Zustände in anderen Paketen befinden, bleibt nichts anderes als der Zugriffrechtstyp `public` übrig. Es wäre damit allerdings auch möglich, diese Methoden vom Hauptprogramm direkt aufzurufen und damit den Apportierhamster gewissermaßen zu „versklaven“, da der dann überhaupt nicht mehr entscheiden könnte, was er tut.
Einschub: In C++ gäbe es für dieses Problem übrigens eine elegante Lösung. Man kann hier so genannte `friend`-Funktionen festlegen: Man würde also jene Methoden als `private` belassen, `HamsterZustand` jedoch zur `friend`-Klasse von `ApportierHamster` erklären, die als solche auch `private`-Methoden von außen ausnahmsweise aufrufen darf. An diesem Beispiel sieht man einen der wesentlichen Unterschiede zwischen Java und C++: Java versucht, konsequent die Kapselung der Klassen zu schützen, aber gerade in unserem Fall wird die Kapselung durch diesen konsequenten Umgang mit den Zugriffsrechten geschwächt. C++ bietet eine Ausnahmeregelung an und würde zwar in unserem Fall dadurch die Kapselung verstärken, aber es wird auch damit die Möglichkeit gegeben, die Kapselung zu verletzen, und gerade unerfahrene Programmierer laufen Gefahr, solche Fallen zu übersehen. Will nämlich ein Programmierer die Kapselung seiner Methoden überprüfen, genügt es in Java, darauf zu achten, ob sie `private` sind oder nicht. In C++ muss der Programmierer für alle `private`-Methoden `friend`-Klassen suchen und diese müssen wiederum untersucht werden, wem sie Zugriff auf die besagten Methoden gestatten.[18]
Zurück zum Apportierhamster: Hier die neue `HamsterZustand`-Klasse:

```
package entwurfsmuster.zustand.zustandC;

abstract public class HamsterZustand extends Hamster {

    public static void aendereZustand(ApportierHamster hamster,
                                      HamsterZustand zustand) {
        hamster.aendereZustand(zustand);
    }
```

[17] In [BB04], S.430 (Kapitel 14.1 Zugriffsrechte) auch „<ohne>-Zugriffsrecht“ genannt)
[18] vgl. zum Thema friend-Klassen z.B. [Bre05], S.386ff (Kapitel 11.4.1)

```
    abstract public void streichle(ApportierHamster hamster);
    abstract public void wecke(ApportierHamster hamster);
    abstract public void lasseApportieren(ApportierHamster hamster,
                                          int reihe, int spalte);
    abstract public void fuettere(ApportierHamster hamster,
                                  int reihe, int spalte);
}
```

Nun können wir unseren ersten konkreten Hamsterzustand implementieren. Wir haben jetzt alle Probleme gelöst, die uns unterwegs begegnet sind, und der einzige Grund, warum der folgende Quellcode noch nicht vollständig kompiliert, ist, dass Zustandsänderungen noch nicht möglich sind, weil wir andere Zustände noch nicht implementiert haben.

```
package entwurfsmuster.zustand.zustandC;

public class SchlaeftSattZufrieden extends HamsterZustand {

    private static SchlaeftSattZufrieden exemplar;

    public static HamsterZustand exemplar() {
        if (exemplar == null) {
            exemplar = new SchlaeftSattZufrieden();
        }
        return exemplar;
    }

    public void streichle(ApportierHamster hamster) {
        hamster.schreib("***keine Reaktion***");
    }

    public void wecke(ApportierHamster hamster) {
        hamster.aufwachen();
        hamster.aendereZustand(SattZufrieden.exemplar());
    }

    public void lasseApportieren(ApportierHamster hamster,
                                 int reihe, int spalte) {
        hamster.schreib("***keine Reaktion***");
    }

    public void fuettere(ApportierHamster hamster, int reihe, int spalte) {
        hamster.schreib("***keine Reaktion***");
    }
}
```

Die Überschrift dieses Abschnitts beschreibt den entscheidenden Punkt des Zustandsmusters: Doppelte Delegation. Was damit gemeint ist, will ich an einem Fallbeispiel demonstrieren. Nehmen wir an, der Apportierhamster wird gefüttert und befindet sich in einem Zustand, in dem er nicht satt ist:

1. Die Methode `fuettere` des `ApportierHamster` wird aufgerufen.
2. Diese *delegiert* das Füttern an den aktuellen Zustand.
3. Dieser ruft die `fresse`-Methode des `ApportierHamster` auf...
4. ...und delegiert dann die Zustandsänderung wieder an den Hamster.

3.5.6 Anwendung des Zustands-Muster

Was noch zu tun bleibt, ist die Implementierung der anderen konkreten Zustände. Um diese zu bestimmen und sinnvolle Zustandsübergänge zu ermitteln, ist eine Visualisierung sinnvoll – nahe liegt, sich der UML-Syntax für Zustandsdiagramme zu bedienen.

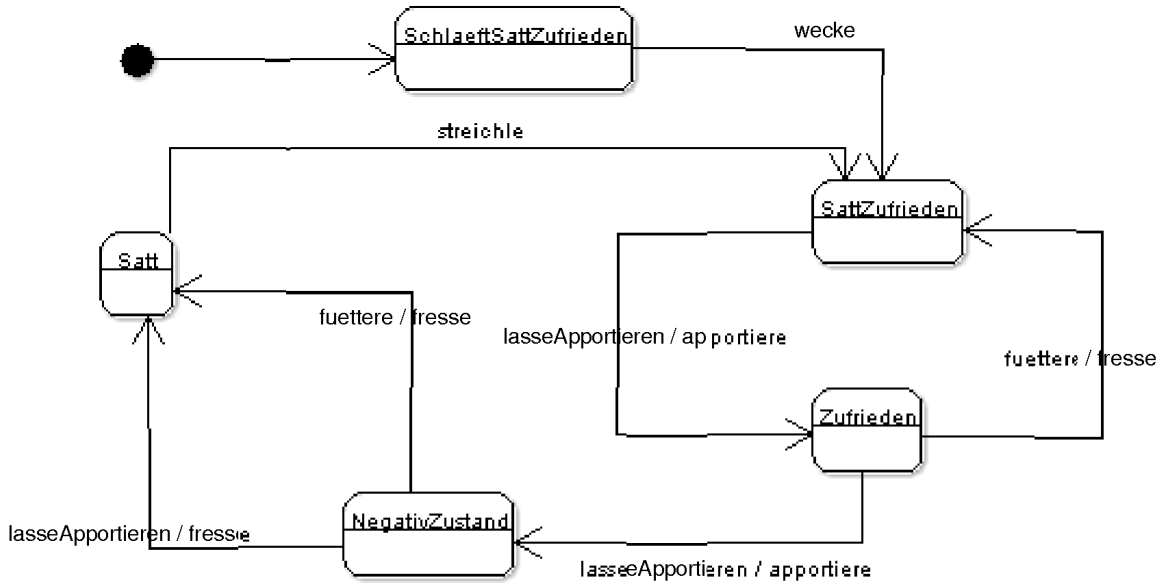


Abbildung 6: Die Zustände des Apportierhamsters als UML-Zustandsdiagramm

Abbildung 6 stellt das Verhalten des Apportierhamsters aus unserer ersten Version dar. Die abgerundeten Rechtecke stehen für Zustände, in denen sich ein Objekt (in unserem Fall also unser ApportierHamster) befinden kann, und die Kanten (bzw. deren Beschriftung vor dem Schrägstrich) stellen Methoden dar, mit denen Zustandsübergänge ausgelöst werden. Falls ein Verhalten dem Hamster gegenüber ein Verhalten seinerseits nach sich zieht, wird dies durch eine entsprechende Methodenbezeichnung nach dem Schrägstrich ausgedrückt.
Übrigens musste ich bei einer Klasse von meiner Namenskonvention für konkrete Zustandsklassen abweichen: Die Klasse, die den Zustand „schlaeft nicht, nicht satt, nicht zufrieden“ repräsentiert, heißt `NegativZustand`.
Wir können nun leicht ablesen, welche konkreten Zustandsklassen uns noch fehlen und diese implementieren. Wie bei der ersten wird bei allen anderen ebenfalls das Singleton-Muster eingesetzt.

```
package entwurfsmuster.zustand.zustandC;
```

```
public class SattZufrieden extends HamsterZustand {

    private static SattZufrieden exemplar;

    public static HamsterZustand exemplar() {
        if (exemplar == null) {
            exemplar = new SattZufrieden();
        }
        return exemplar;
    }

    public void streichle(ApportierHamster hamster) {
        hamster.schreib("***Schwanz wedeln***");
    }

    public void wecke(ApportierHamster hamster) {
        hamster.schreib("***keine Reaktion***");
    }

    public void lasseApportieren(ApportierHamster hamster,
                                 int reihe, int spalte) {
        hamster.apportiere(reihe, spalte);
        hamster.aendereZustand(Zufrieden.exemplar());
    }

    public void fuettere(ApportierHamster hamster, int reihe, int spalte) {
        hamster.fresse(reihe, spalte);
    }
}

package entwurfsmuster.zustand.zustandC;

public class Zufrieden extends HamsterZustand {
    private static Zufrieden exemplar;

    public static HamsterZustand exemplar() {
        if (exemplar == null) {
            exemplar = new Zufrieden();
        }
        return exemplar;
    }

    public void streichle(ApportierHamster hamster) {
        hamster.schreib("***hungriger Blick***");
    }

    public void wecke(ApportierHamster hamster) {
        hamster.schreib("***keine Reaktion***");
```

```
    }

    public void lasseApportieren(ApportierHamster hamster,
                                 int reihe, int spalte) {
        hamster.apportiere(reihe, spalte);
        hamster.aendereZustand(NegativZustand.exemplar());
    }

    public void fuettere(ApportierHamster hamster, int reihe, int spalte) {
        hamster.fresse(reihe, spalte);
        hamster.aendereZustand(SattZufrieden.exemplar());
    }
}
```

```
package entwurfsmuster.zustand.zustandC;

public class Satt extends HamsterZustand {

    private static Satt exemplar;

    public static HamsterZustand exemplar() {
        if (exemplar == null) {
            exemplar = new Satt();
        }
        return exemplar;
    }

    public void streichle(ApportierHamster hamster) {
        hamster.schreib("***Schwanz wedeln***");
        hamster.aendereZustand(SattZufrieden.exemplar());
    }

    public void wecke(ApportierHamster hamster) {
        hamster.schreib("***keine Reaktion***");
    }

    public void lasseApportieren(ApportierHamster hamster,
                                 int reihe, int spalte) {
        hamster.schreib("***unwilliger Blick***");
    }

    public void fuettere(ApportierHamster hamster, int reihe, int spalte) {
        hamster.fresse(reihe, spalte);
    }
}
```

```
package entwurfsmuster.zustand.zustandC;

public class NegativZustand extends HamsterZustand {
```

```
    private static NegativZustand exemplar;

    public static HamsterZustand exemplar() {
        if (exemplar == null) {
            exemplar = new NegativZustand();
        }
        return exemplar;
    }

    public void streichle(ApportierHamster hamster) {
        hamster.schreib("***beißt in Deine Hand!***");
    }

    public void wecke(ApportierHamster hamster) {
        hamster.schreib("***keine Reaktion***");
    }

    public void lasseApportieren(ApportierHamster hamster,
                                 int reihe, int spalte) {
        hamster.fresse(reihe, spalte);
        hamster.aendereZustand(Satt.exemplar());
    }

    public void fuettere(ApportierHamster hamster, int reihe, int spalte) {
        hamster.fresse(reihe, spalte);
        hamster.aendereZustand(Satt.exemplar());
    }
}
```

Das alte Hauptprogramm kann unverändert verwendet weiter verwendet werden.

3.5.7 Zusammenfassung und Ausblick

Das Entwurfsmuster Zustand ist ein objektbasiertes Verhaltensmuster. Abbildung 7 zeigt ein Klassendiagramm, das abstrakt die wesentlichen Merkmale des Musters zusammenfasst.
Wir können Situationen, in denen dieses Entwurfmuster angewandt werden kann, daran erkennen, ob das verschiedene Verhalten eines Objektes in verschiedenen Zuständen durch einen großen Block an Bedingungsanweisungen dargestellt wird (oder werden kann).
Ebenso können wir versuchen, das Verhalten dieses Objektes als Zustandsdiagramm auszudrücken. Gelingt dies, ist das ebenfalls ein Zeichen für die Anwendbarkeit der hier vorgestellten Verfahrensweise.
Man kann also sagen, dass wenn ein Problem vom Entwickler am intuitivsten in Form von in einander übergehenden Zuständen gedacht werden kann, das Entwurfsmuster Zustand eine Möglichkeit gibt, die Klassenstruktur nach diesem System aufzubauen.

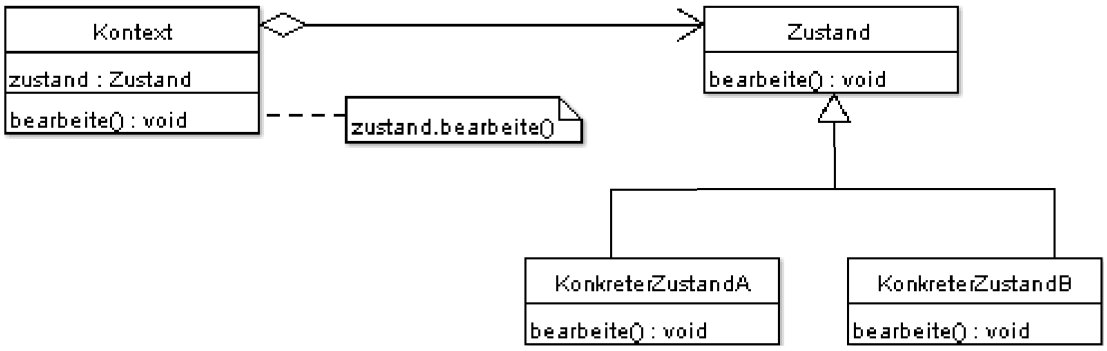


Abbildung 7: Klassendiagramm zum Entwurfsmuster Zustand

3.6 Vermittler (Mediator)

3.6.1 Motivationsaufgabe

Im Territorium gibt es ein kleines Labyrinth und ein Körnerfeld. Es gibt eine Hamsterfamilie, die durch folgende Klassen repräsentiert werden soll:

- Ein `VaterHamster`, dem das Territorium „gehört“. Er kennt das Labyrinth auswendig und weiß, wie man Körner „anbaut“.
- Ein `FaulerSohnHamster`, der Körner vom Feld frisst. Er ist gerade so anständig, dass er den Vater vorher fragt, ob er ein Korn fressen darf.
- Ein `DienerHamster`, der dem Vater meldet, wieviele Körner auf dem Feld liegen und von diesem Anweisungen bekommt, wann wieder Körner auf dem Feld „nachzupflanzen“ sind, damit immer genug da ist.
- Ein `FleissigerSohnHamster`, der versucht, sich die Wege des Labyrinths einzuprägen. Immer wenn er an eine Abzweigung kommt, fragt er den Vater, in welche Richtung er gehen muss. Wenn er durch das Labyrinth hindurch ist, verbietet der Vater dem faulen Sohn, weiter zu fressen, damit der fleißige Sohn auch etwas bekommt. Der ist nun nach dem Trainieren sehr hungrig.

3.6.2 Zum Territorium

Das Territorium betreffend muss ich auf folgenden Umstand hinweisen: Ich sah mich vor die Wahl gestellt, die Hamsterfamilie auf ein ganz bestimmtes Territorium zuzuschneiden mit genauen Startpositionen für die einzelnen Hamster – oder den Programmieraufwand und die Länge der Klassen enorm anwachsen zu lassen, um eine relative Unabhängigkeit dieses Beispiels vom Territorium zu erreichen.
Ich habe mich für die erste Möglichkeit entschieden, da hier im Vordergrund die Erläuterung des Entwurfsmusters steht und ich – mit dem Ziel der besseren Verständlichkeit – Dinge, die den Quellcode unleserlicher machen, wie festimplementierte Zahlen, in Kauf nehmen muss.
In Abbildung 8 ist das Territorium zu sehen, das zu dem im Folgenden erläuterten Quellcode passt.

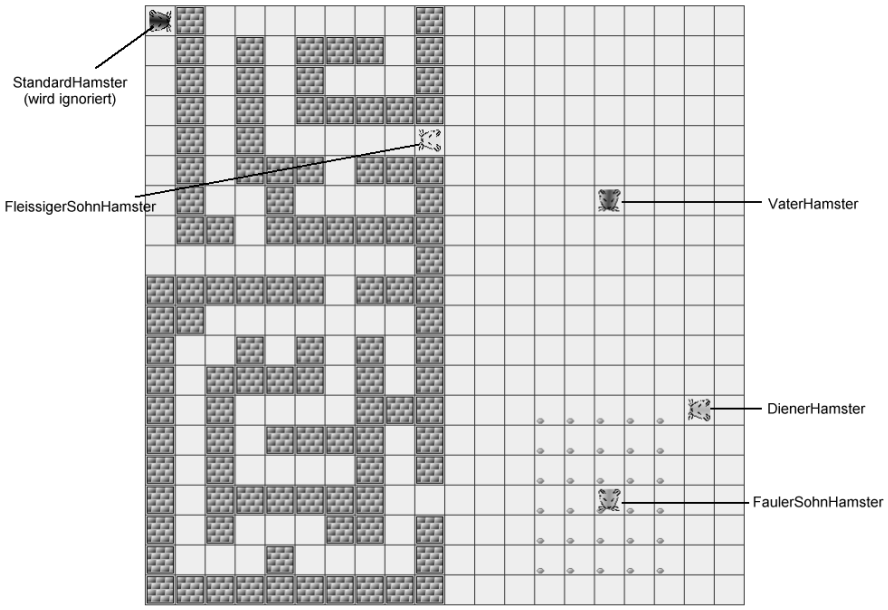


Abbildung 8: Hamsterterritorium zum Vermittler-Beispiel

3.6.3 Die Klassen für das Muster

Basisbestandteile der hier vorgestellten Implementierung des Vermittler-Musters sind das Interface `DirektorHamster` und die abstrakte Klasse `ElementHamster`. Indem nun eine Klasse `DirektorHamster` implementiert und alle anderen von `ElementHamster` erben, wird eine Hierarchie festgelegt, in der einer bestimmt, was gemacht wird – in unserem Beispiel der `VaterHamster` – und alle anderen diesem melden, was passiert und Anweisungen entgegennehmen. Insofern ist also der offizielle Name dieses Musters – Vermittler – etwas missverständlich, da er impliziert, es handle sich um eine Kommunikation auf gleicher Ebene, die von einem Vermittler ermöglicht und moderiert wird.
„Direktoren“ müssen zwei Methoden implementieren:

- Eine Methode, die die untergeordneten Objekte erzeugt, hier `erzeugeElementHamster`.
- Eine `behandleEreignis`-Methode, die in Abhängigkeit vom `ElementHamster` entscheidet, was zu tun ist.

```
package entwurfsmuster.vermittler;

public interface DirektorHamster {
    public void behandleEreignis(ElementHamster element);
    public void erzeugeElementHamster();
}
```

Ein Element kennt seinen Direktor (und speichert ihn als Instanzattribut). Außerdem hat er eine Methode `behandleEreignis`, die `behandleEreignis` des Direktors aufruft. Diese Delegation realisiert die Forderung, ein Element solle nur melden, dass ein Ereignis eingetreten ist, der Direktor es aber behandeln soll.

```
package entwurfsmuster.vermittler;
```

```
import entwurfsmuster.AllroundHamster;

abstract public class ElementHamster extends AllroundHamster {

    private DirektorHamster direktor;

    public ElementHamster(int r, int s, int b, int k,
                          DirektorHamster direktor) {
        super(r, s, b, k);
        this.direktor = direktor;
    }

    public void behandleEreignis() {
        direktor.behandleEreignis(this);
    }
}
```

Der fleißige Sohn hat seine Startposition am Anfang des Labyrinths und geht solange selbstständig hindurch, bis er an eine Abzweigung kommt, d.h. bis es mehrere Möglichkeiten zum Weitergehen gibt. Dann fragt er den Vater, in welche Richtung er gehen muss. Übrigens steht eine Ausführung der dafür verantwortlichen Methode `erforscheLabyrinth` nur für einen Schritt des Hamsters, damit die Gleichzeitigkeit der Aktivitäten aller Hamster simuliert werden kann: In der Klasse `VaterHamster` gibt es eine Methode `steuereEreignisse`, die in einer Schleife diese „Aktivitätsmethoden“ der anderen Hamster aufruft.

```
package entwurfsmuster.vermittler;

public class FleissigerSohnHamster extends ElementHamster {

    public FleissigerSohnHamster(DirektorHamster direktor) {
        super(4, 9, WEST, 0, direktor);
    }

    public void erforscheLabyrinth() {
        int wege = 0;
        if (vornFrei()) {
            wege++;
        }
        if (linksFrei()) {
            wege++;
        }
        if (rechtsFrei()) {
            wege++;
        }
        if (wege > 1) {
            schreib("Fleissiger Sohn: Hier ist eine Abzweigung. "
                        + "Vater, wo muss ich lang?");
            behandleEreignis();
```

```
        } else {
            if (vornFrei()) {
                vor();
            } else if (linksFrei()) {
                linksUm();
                vor();
            } else if (rechtsFrei()) {
                rechtsUm();
                vor();
            } else {
                schreib("Fleissiger Sohn: Ich habe mich verlaufen!");
            }
        }
    }

    public void fresseKoerner() {
        int schritte = 5;
        gotoKachel(13, 18);
        setzeBlickrichtung(WEST);
        for (int i = 0; i < 10; i++) {
            for (int j = 0; j < schritte; j++) {
                vor();
                if (kornDa()) {
                    nimm();
                }
            }
            linksUm();
            if (i % 2 != 0) {
                schritte--;
            }
        }
    }
}
```

Der faule Sohn bewegt sich mit zufällig gewählter Richtung im Bereich des Hamsterfeldes. Gelegentlich bekommt er Hunger und fragt den Vater, ob er ein Korn fressen darf:

```
package entwurfsmuster.vermittler;

public class FaulerSohnHamster extends ElementHamster {

    public FaulerSohnHamster(DirektorHamster direktor) {
        super(16, 15, SUED, 0, direktor);
    }

    public void spaziere() {
        int zufallRichtung = (int) (Math.random() * 4);
        int zufallHunger = (int) (Math.random() * 3);
        if (pruefeFeld(zufallRichtung)) {
```

```
            setzeBlickrichtung(zufallRichtung);
            vor();
            if (kornDa() && zufallHunger == 0) {
                schreib("Fauler Sohn: Vater, da liegt ein Korn. "
                            + "Darf ich es fressen?");
                behandleEreignis();
            }
        }
    }

    private boolean pruefeFeld(int richtung) {
        switch (richtung) {
        case NORD:
            return getReihe() > 12;
        case OST:
            return getSpalte() < 17;
        case SUED:
            return getReihe() < 18;
        case WEST:
            return getSpalte() > 13;
        }
        return false;
    }
}
```

Der Diener kann zählen, wie viele Körner auf dem Feld sind und Körner nachpflanzen.

```
package entwurfsmuster.vermittler;

public class DienerHamster extends ElementHamster {

    public DienerHamster(DirektorHamster direktor) {
        super(13, 18, WEST, 100, direktor);
    }

    public void zaehleKoernerAufFeld() {
        int anzahl = 0;
        for (int i = 13; i <= 18; i++) {
            for (int j = 13; j <= 17; j++) {
                anzahl += Territorium.getAnzahlKoerner(i, j);
            }
        }
        if (anzahl < 30) {
            schreib("Auf dem Feld liegen weniger als 30 Koerner.");
            behandleEreignis();
        }
    }

    public void pflanzeKoernerAn() {
```

```
        int schritte = 5;
        for (int i = 0; i < 10; i++) {
            for (int j = 0; j < schritte; j++) {
                int zufall = (int) (Math.random() * 2);
                vor();
                if (zufall == 0) {
                    gib();
                }
            }
            linksUm();
            if (i % 2 != 0) {
                schritte--;
            }
        }
        gotoKachel(13,18);
        setzeBlickrichtung(WEST);
    }
}
```

Die Klasse `VaterHamster` hat über alles die Kontrolle:

- Mit `erzeugeElementHamster` schafft sie Instanzen für die drei anderen Hamster.
- Mit `behandleEreignis` nimmt sie die Meldungen der anderen Hamster entgegen und entscheidet, was zu tun ist:
  - Dem fleißigen Sohn wird die richtige Richtung durchs Labyrinth gewiesen.
  - Dem faulen Sohn wird gestattet oder verboten, Körner zu fressen.
  - Dem Diener wird befohlen, Körner nachzupflanzen.
- Mit `steuereEreignisse` wird – wie schon erwähnt – die Gleichzeitigkeit der verschiedenen Aktivitäten simuliert. Sie gehört übrigens nicht zum Vermittlermuster, sondern wurde von mir angefügt.

```
package entwurfsmuster.vermittler;

public class VaterHamster extends Hamster implements DirektorHamster {

    public final static int[][] labyrinth = { { 4, 6, WEST}, { 0, 4, WEST},
                                              { 8, 3, OST }, { 8, 6, SUED},
                                              {10, 6, WEST}, {10, 4, WEST},
                                              {16, 8, OST }                };

    private FaulerSohnHamster faulerSohn;
    private FleissigerSohnHamster fleissigerSohn;
    private DienerHamster diener;

    public VaterHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
```

```
}

public void erzeugeElementHamster() {
    this.faulerSohn = new FaulerSohnHamster(this);
    this.fleissigerSohn = new FleissigerSohnHamster(this);
    this.diener = new DienerHamster(this);
}

public void behandleEreignis(ElementHamster element) {
    if (element.equals(fleissigerSohn)) {
        for (int[] koordinaten : labyrinth) {
            if (koordinaten[0] == element.getReihe()
                    && koordinaten[1] == element.getSpalte()) {
                element.setzeBlickrichtung(koordinaten[3]);
                switch (koordinaten[3]) {
                case NORD:
                    schreib("Vater: Du musst nach Norden gehen.");
                    break;
                case OST:
                    schreib("Vater: Du musst nach Osten gehen.");
                    break;
                case SUED:
                    schreib("Vater: Du musst nach Sueden gehen.");
                    break;
                case WEST:
                    schreib("Vater: Du musst nach Westen gehen.");
                    break;
                }
                break;
            }
        }
    } else if (element.equals(faulerSohn)) {
        int zufallszahl = (int) (Math.random() * 2);
        if (zufallszahl == 0) {
            schreib("Vater: Du darfst das Korn fressen.");
            faulerSohn.nimm();
        } else {
            schreib("Vater: Du darfst das Korn nicht fressen.");
        }
    } else if (element.equals(diener)) {
      schreib("Das sind zu wenig. Pflanze Koerner nach.");
      diener.pflanzeKoernerAn();
    }
}

public void steuereAblauf() {
    int zaehler = 0;
    while(fleissigerSohn.getReihe() !=  16
```

```
                || fleissigerSohn.getSpalte() != 10) {
            fleissigerSohn.erforscheLabyrinth();
            faulerSohn.spaziere();
            if (zaehler % 5 == 0) {
                diener.zaehleKoernerAufFeld();
            }
            zaehler++;
        }
        schreib("Vater: Fauler Sohn, Du hast genug gefressen, "
                    + "jetzt ist Dein fleissiger Bruder dran.");
        fleissigerSohn.fresseKoerner();
    }
}
```

Im Hauptprogramm wird zunächst der `VaterHamster` erzeugt. Dieser erzeugt dann seine `ElementHamster` und ruft dann die Methode `steuereAblauf` auf:

```
import entwurfsmuster.vermittler.VaterHamster;

void main() {
    VaterHamster vater = new VaterHamster(6, 15, Hamster.SUED, 0);

    vater.erzeugeElementHamster();
    vater.steuereAblauf();
}
```

3.6.4 Zusammenfassung und Ausblick

Das Entwurfsmuster Vermittler ist ein objektbasiertes Verhaltensmuster. Abbildung 9 zeigt ein Klassendiagramm, das abstrakt die wesentlichen Merkmale des Musters zusammenfasst.

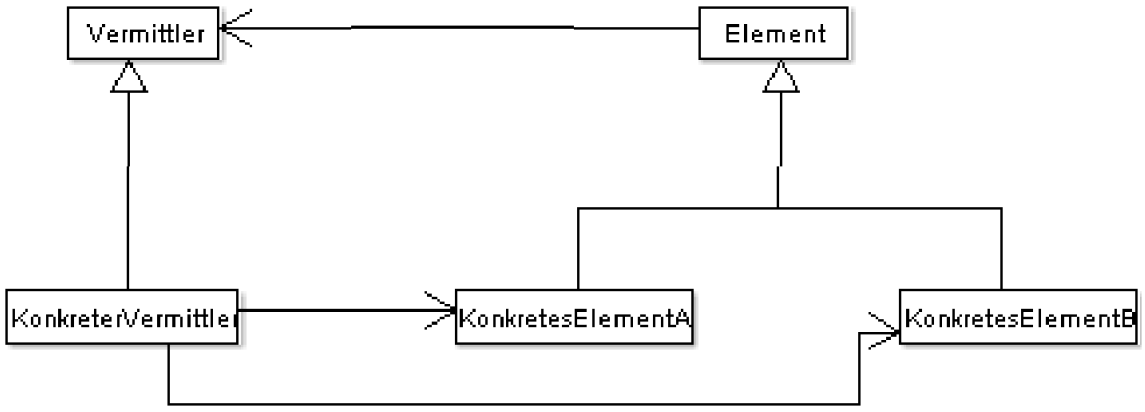


Abbildung 9: Klassendiagramm zum Entwurfsmuster Vermittler

In [GHJV] wird als Beispiel eine Dialogbox mit verschiedenen Elementen (Schaltflächen, Eingabefelder usw) verwendet. Diese Elemente melden einem Dialogdirektor, wenn bei ihnen ein

Ereignis auftritt (z.B. Schaltfläche wurde geklickt). Der Dialogdirektor entscheidet dann, was geschehen soll. Dies hat den Vorteil, dass nicht jedes beliebige Element Entscheidungen über die ganze Dialogbox treffen kann, sondern es eine zentrale Stelle gibt, wo dies passiert.
Deswegen wurde auch im Hamsterprogramm dieses Kapitels der Namen `ElementHamster` gewählt. Er verdeutlicht die uneigenständige, untergeordnete Rolle dieser Hamster.

Man könnte auf die Idee kommen, dass bei diesem Muster radikal die Kapselung der Elementklassen zerstört wird: Der Direktor entscheidet ja fortwährend, was diese ganz konkret tun sollen. Allerdings zeigt auch das Dialogbeispiel, dass gerade dies sinnvoll sein kann, Elementklassen wirklich als Unterklassen zu verstehen und der Gedanke der Kapselung hier untergeordnet werden muss.

3.7 Zuständigkeitskette (Chain of Responsibility)

3.7.1 Motivationsaufgabe

Das Territorium ist in vier Abschnitte unterteilt, und in jedem Abschnitt befinden sich ein paar Körner und ein Hamster. Ein weiterer Hamster befindet sich in der Mitte des Territoriums und fragt vom Benutzer ab, in welchem der vier Abschnitte ein Korn gesammelt werden soll. Der Hamster in der Mitte kennt aber nur den ersten der SammelHamster und weist ihm die Aufgabe zu. Ist sein Abschnitt der gewünschte, übernimmt er die Aufgabe; ist er es jedoch nicht, reicht er die Aufgabe an den nächsten Hamster weiter. Dieser und die folgenden verfahren ebenso.

3.7.2 Territorium

Wie schon in der Aufgabenstellung angedeutet, soll das Territorium mit Mauern in vier Abschnitte eingeteilt werden. Im mitgelieferten Beispielterritorium (siehe Abbildung 10) wird ebenfalls durch Setzung der Mauern die Anweisung umgesetzt, der mittlere Hamster solle nur einen der SammelHamster kennen.

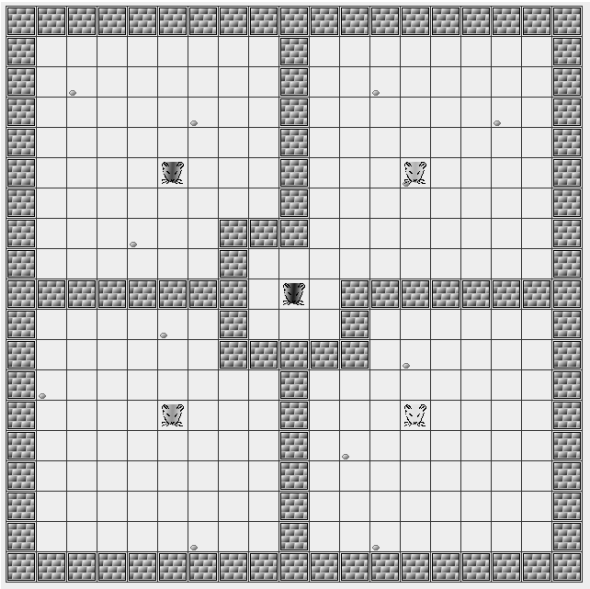


Abbildung 10: Passendes Territorium zum Hamsterprogramm zur Zuständigkeitskette

3.7.3 Der AnfrageHamster

Als erstes müssen wir die geforderte Kette von Hamstern bilden, d.h. jeder Hamster muss seinen Nachfolger kennen, an den er eine Anfrage weiterleiten kann. Dazu implementieren wir eine abstrakte Klasse `AnfrageHamster`, in der dieses Prinzip verankert ist. Mit der Methode `bearbeiteAnfrage` kann eine Anfrage an den nachfolgenden Hamster in der Kette delegiert werden. Eine Anfrage wird über einen Parameter von Typ `Object` definiert, damit der `AnfrageHamster` möglichst allgemein wiederverwendbar ist.
Dies ist die Basisklasse des Entwurfsmusters **Zuständigkeitskette**.

```
package entwurfsmuster.zustaendigkeitskette;
```

```
import entwurfsmuster.AllroundHamster;

abstract public class AnfrageHamster extends AllroundHamster {

    private AnfrageHamster nachfolger;

    public AnfrageHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }

    public void meldeAn(AnfrageHamster nachfolger) {
        this.nachfolger = nachfolger;
    }

    public void bearbeiteAnfrage(Object anfrage) {
        if (this.nachfolger != null) {
            this.nachfolger.bearbeiteAnfrage(anfrage);
        }
    }
}
```

`FeldHamster` ist nun die konkrete Klasse, in der die eigentliche Aufgabe (das Sammeln eines Korns) umgesetzt werden soll. Die geerbte Methode `bearbeiteAnfrage` wird überschrieben. Sie überprüft nun, ob die Anfrage einen `int`-Wert beinhaltet und ob das gegebene Feld das eigene ist und leitet daraufhin die Anfrage weiter oder bearbeitet sie selbst (und ruft die Methode `sammle` auf).[19]
Der Sammelalgorithmus wurde der Einfachheit halber vom `SammelHamster` aus dem Kapitel zum Beobachter-Muster entnommen (siehe Kapitel 3.1, Seite 22).

```
package entwurfsmuster.zustaendigkeitskette;

public class FeldHamster extends AnfrageHamster {

    private int feld;

    public FeldHamster(int r, int s, int b, int k, int feld) {
        super(r, s, b, k);
        this.feld = feld;
    }

    public void bearbeiteAnfrage(Object anfrage) {
        schreib("FeldHamster " + this.feld + " bearbeitet Anfrage.");
        if (anfrage instanceof Integer) {
            int feld = (Integer) anfrage;
            if (this.feld == feld) {
                schreib("FeldHamster " + this.feld + " übernimmt Anfrage.");
```

[19] Zur Wiederholung empfehle ich [BB044], S.167 (Kapitel 7.4.2 Zugriff auf geerbte überschriebene Methoden)

```
                sammle();
            } else {
                super.bearbeiteAnfrage(anfrage);
            }
        }
    }

    public void sammle() {
        while (getAnzahlKoerner() < 1) {
            int zufall = (int) (Math.random() * 4);
            switch (zufall) {
            case 0:
                vorUndNimm();
                break;
            case 1:
                linksUm();
                vorUndNimm();
                break;
            case 2:
                rechtsUm();
                vorUndNimm();
                break;
            case 3:
                kehrt();
                vorUndNimm();
            }
        }
    }
}
```

In der Literatur wird ein Objekt, welches Funktionalitäten anderer Objekte benutzt, üblicherweise als „Klient“ bezeichnet[20] . Dementsprechend bezeichnen wir den Hamster, der unsere Hamsterkette benutzt, als `KlientHamster`. Der Aufgabe entsprechend kennt er nur ein Kettenglied – dies wird mit einen Instanzattribut realisiert, das (wie üblich) im Konstruktor initialisiert wird.

```
package entwurfsmuster.zustaendigkeitskette;

public class KlientHamster extends Hamster {

    private AnfrageHamster hamster;

    public KlientHamster(int r, int s, int b, int k,
                         AnfrageHamster hamster) {
        super(r, s, b, k);
        this.hamster = hamster;
    }
```

[20] vlg. z.B. [GHJV04], S. 110, 174, 278

```
    public void delegiereAnfrage() {
        Object wahlFeld =
                liesZahl("Auf welchem Feld soll gesammelt werden?");
        this.hamster.bearbeiteAnfrage(wahlFeld);
    }
}
```

Bevor wir uns das Hauptprogramm ansehen, möchte ich noch einmal auf den Umstand hinweisen, dass wir mit nummerierten Feldern arbeiten, dementsprechend auch in Anfragen eine Zahl (die ein Feld bezeichnet) weitergeben. Um eine größere Übersicht zu gewährleisten, bietet es sich an, für die vier Felder Konstanten `FELD_1`, `FELD_2`, `FELD_3` und `FELD_4` zu definieren usw.[21]
Es gibt mehrere Möglichkeiten, wo man im Quellcode diese Konstantendefinition vornehmen kann. Eine davon ist, auszunutzen, dass in Interfaces Konstanten definiert werden können, die auch von den Klassen genutzt werden können, die diese Interfaces implementieren. Der Vorteil dieser Vorgehensweise ist, dass man alle häufig genutzten Konstanten in einem großen Klassenverbund an einer Stelle sammeln kann – und jede Klasse, die auf die Konstanten zugreifen soll, muss nur besagtes Interface implementieren. In unserem Fall würde das dann folgendermaßen aussehen:

```
package entwurfsmuster.zustaendigkeitskette;

public interface FeldKonstanten {
    public static final int FELD_1 = 1;
    public static final int FELD_2 = 2;
    public static final int FELD_3 = 3;
    public static final int FELD_4 = 4;
}
```

Da wir jedoch im Hamstersimulator das Hauptprogramm nicht in einer Klasse eingebettet definieren, haben wir in diesem Beispiel nicht die Möglichkeit, mit `implements FeldKonstanten` den Zugriff auf die Konstanten zu gewährleisten. Dieses Problem können wir mit einem neuen Feature der Java-Version 5.0 lösen: dem statischen Import. Hiermit kann man die statischen Elemente (Attribute und Methoden) einer Klasse (oder eben Konstanten eines Interface) mit einer einzigen Anweisung importieren.

```
import entwurfsmuster.zustaendigkeitskette.FeldHamster;
import entwurfsmuster.zustaendigkeitskette.KlientHamster;
import static entwurfsmuster.zustaendigkeitskette.FeldKonstanten.*;

void main() {
    FeldHamster feldHamster1 =
            new FeldHamster(5, 13, Hamster.SUED, 0, FELD_1);
    FeldHamster feldHamster2 =
            new FeldHamster(13, 13, Hamster.SUED, 0, FELD_2);
    FeldHamster feldHamster3 =
```

[21] Zur Wiederholung empfehle ich [BB04], S.139 (Kapitel 6.8.3 Vorteile von Konstanten)

```
            new FeldHamster(13, 5, Hamster.SUED, 0, FELD_3);
    FeldHamster feldHamster4 =
            new FeldHamster(5, 5, Hamster.SUED, 0, FELD_4);

    feldHamster1.meldeAn(feldHamster2);
    feldHamster2.meldeAn(feldHamster3);
    feldHamster3.meldeAn(feldHamster4);
    feldHamster4.meldeAn(feldHamster1);

    KlientHamster klient =
            new KlientHamster(9, 9, Hamster.SUED, 0, feldHamster1);

    klient.delegiereAnfrage();
}
```

3.7.4 Zusammenfassung und Ausblick

Das Entwurfsmuster Zuständigkeitskette ist ein objektbasiertes Verhaltensmuster. Abbildung 11 zeigt ein Klassendiagramm, das abstrakt die wesentlichen Merkmale des Musters zusammenfasst.

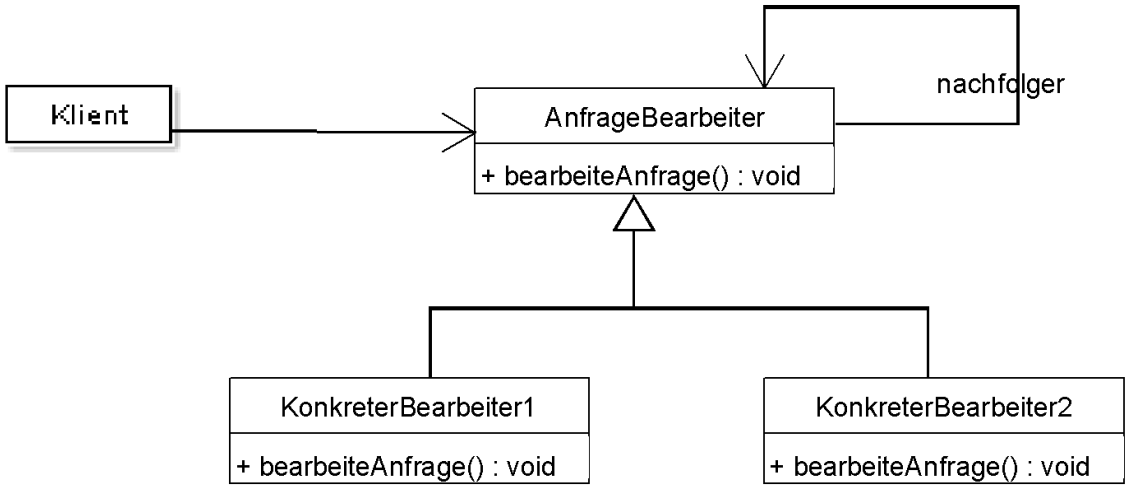


Abbildung 11: Klassendiagramm zum Entwurfsmuster Zuständigkeitskette

Eine der wichtigsten Entscheidungen, die man bei der Implementierung einer Zuständigkeitskette zu treffen hat, ist, ob man eine *geschlossene* oder eine *offene* Kette nutzen will.

- Eine geschlossene Kette hat den Vorteil, dass der Einstiegspunkt keine Rolle spielt bzw. keine Hierarchie der Kettenglieder notwendig ist.

  Unbedingt zu beachten ist jedoch, dass in diesem Fall der Programmierer garantieren muss, dass sich auf jeden Fall ein Kettenglied für die Abarbeitung der Anfrage zuständig fühlt, weil sonst die Gefahr einer Endlosschleife besteht, in der eine Anfrage immer weitergereicht wird, ohne je bearbeitet zu werden.

- Bei einer offenen Kette ist nicht mehr gleichgültig, welches Kettenglied den Auftrag vom Klienten bekommt. Nur, wenn es das erste ist, besteht die Möglichkeit, dass der Auftrag die gesamte Kette durchläuft.

  Die Gefahr einer Endlossschleife ist hier zwar nicht mehr gegeben – aber auch hier gibt es keine Abarbeitungsgarantie. Wenn sich das letzte Element in der Kette nicht zuständig fühlt, fällt die Anfrage sozusagen ins Nichts.

Im vorgestellten Beispiel wird eine geschlossene Kette verwendet, da garantiert werden kann, dass ein Kettenglied den Auftrag annimmt.

3.8 Interpreter

Wenn wir Menschen einen Text lesen und verstehen, passieren im Grunde mehrere Dinge gleichzeitig:

1. Wir führen eine lexikalische Analyse durch: Wir erkennen die einzelnen Zeichen als Bestandteile des uns bekannten Alphabets.

2. Wir führen eine syntaktische Analyse durch: Wir erkennen Wörter und Satzstrukturen.

3. Wir führen eine semantische Analyse durch: Wir erkennen die Bedeutung einzelner Wörter und ganzer Satzzusammenhänge und können schließlich auf den Inhalt schließen, auf das, was der Text uns sagen will.

Ein geübter menschlicher Leser nimmt gar nicht mehr wahr, dass er dies alles tut, er „liest“ einfach. Wenn man einen Compiler für eine Sprache schreiben will, so dass der Computer die Sprache „lesen“ d.h. verstehen kann, muss man all dies beachten und getrennt implementieren. Hauptsächlich kommt der Compilerbau natürlich bei der Entwicklung von Programmiersprachen zum Einsatz, jedoch kann es auch bei einem kleinerem Problem sinnvoll sein, „Exemplare dieses Problems als Sätze einer einfachen Sprache auszudrücken“ ([GHJV04], S.319), und einen Interpreter für diese Sprache zu bauen, der dann das Problem durch Auswertung dieser Sätze löst. An einem einfachen Beispiel werden wir diese Art der Problemlösung im folgenden Kapitel durchgehen – sie kann maßgeblich durch den Einsatz des Entwurfsmusters **Interpreter** unterstützt werden.
Details und genauere Bedeutung von Fachwörtern aus dem Bereich des Compilerbaus können in [ASU99] nachgelesen werden.

3.8.1 Motivationsaufgabe

Mehrere Hamster sollen gemeinsam folgende Aufgaben lösen:

1. Es soll vom Benutzer ein mathematischer Term eingelesen werden, in dem die Zeichen „+“, „-“, „(„, „)“ und Buchstaben (letztere als Variablennamen) benutzt werden dürfen.

2. Für jede vorkommende Variable soll eine Belegung eingelesen werden.

3. Es soll eine Auswertung des Terms erfolgen und das Ergebnis bekannt gegeben werden.

3.8.2 Territorium

Da bei der Lösung dieser Aufgabe nichts mit dem Hamstersimulator visualisiert werden kann, gibt es auch keine Anforderungen an das Territorium.

3.8.3 Die Klassen für das Interpreter-Muster

Basis des Interpreter-Musters ist ein Interface, das einen abstrakten Ausdruck repräsentiert, in unserem Fall `Term`. Eine Klasse, die dieses Interface implementiert, muss drei Methoden besitzen:

- `werteAus`: Ein übergebener Kontext weist jeder im Term vorkommenden Variable einen Wert zu, so dass der konkrete Gesamtwert des Terms ausgerechnet werden kann. Dieser wird abschließend zurückgegeben.

- `ersetze`: Mit dieser Methode soll in einem existierenden Term nachträglich eine Variable durch einen Term ersetzt werden können.
- `kopiere`: Eine Hilfsmethode zum Erstellen einer Kopie eines Terms.

```
package entwurfsmuster.interpreter;

public interface Term {
    public int werteAus(Kontext kontext);
    public Term ersetze(String name, Term term);
    public Term kopiere();
}
```

Die Klasse `Kontext` enthält eine `HashMap` und speichert die Belegung der Variablen. Dies wird deswegen in einer gesonderten Klasse gekapselt, damit die Variablen für einen Term flexibel mit anderen Werten belegt werden können bzw. dieselbe Variablenbewegung für mehrere Terme genutzt werden kann.

```
package entwurfsmuster.interpreter;

import java.util.Map;
import java.util.HashMap;

public class Kontext {

    Map<Variable, Integer> map;

    public Kontext() {
        this.map = new HashMap<Variable, Integer>();
    }

    public void weiseZu(Variable variable, int wert) {
        this.map.put(variable, wert);
    }

    public int lookUp(Variable variable) {
        return this.map.get(variable);
    }
}
```

Die einfachste Form eines mathematischen Terms ist eine einzelne Variable.
Die Klasse `Variable` implementiert das Interface `Term` wie folgt:

```
package entwurfsmuster.interpreter;

public class Variable implements Term {

    private String name;
```

```
    public Variable(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public int werteAus(Kontext kontext) {
        return kontext.lookUp(this);
    }

    public Term ersetze(String name, Term term) {
        if (!(name.equals(this.name))) {
            return term.kopiere();
        } else {
            return new Variable(name);
        }
    }

    public Term kopiere() {
        return new Variable(this.name);
    }
}
```

Die beiden anderen Grundterme, die wir benötigen, sind die Addition und die Subtraktion. Sie gleichen sich logischerweise bis auf den Operator. Sie speichern ihre beiden Operanden als Instanzattribute und implementieren die Methoden von `Term` rekursiv, d.h. sie rufen die jeweiligen Methoden ihrer Operanden auf.
Dadurch wird gewährleistet, dass eine Instanz der Klasse `AddTerm` für die Addition zweier Variablen stehen kann, aber auch für die Addition zweier komplexer Terme, da diese erst rekursiv ausgewertet werden und besagte Instanz von `AddTerm` in ihrer `werteAus`-Methode zwei `int`-Werte addiert.

```
package entwurfsmuster.interpreter;

public class AddTerm implements Term {

    private Term op1;
    private Term op2;

    public AddTerm(Term op1, Term op2) {
        this.op1 = op1;
        this.op2 = op2;
    }

    public int werteAus(Kontext kontext) {
        return this.op1.werteAus(kontext) + this.op2.werteAus(kontext);
    }
```

```
    public Term kopiere() {
        return new AddTerm(this.op1.kopiere(), this.op2.kopiere());
    }

    public Term ersetze(String name, Term term) {
        return new AddTerm(this.op1.ersetze(name, term),
                           this.op2.ersetze(name, term));
    }
}
```

```
package entwurfsmuster.interpreter;

public class SubTerm implements Term {

    private Term op1;
    private Term op2;

    public SubTerm(Term op1, Term op2) {
        this.op1 = op1;
        this.op2 = op2;
    }

    public int werteAus(Kontext kontext) {
        return this.op1.werteAus(kontext) - this.op2.werteAus(kontext);
    }

    public Term kopiere() {
        return new SubTerm(this.op1.kopiere(), this.op2.kopiere());
    }

    public Term ersetze(String name, Term term) {
        return new SubTerm(this.op1.ersetze(name, term),
                           this.op2.ersetze(name, term));
    }
}
```

Alle anderen möglichen Terme können mit diesen drei Grundtermen dargestellt werden, da sie beliebig komplex verschachtelt werden können.
Wie der Name schon sagt, ist die Klasse `ParserHamster` dafür zuständig, einen String einzulesen, und mithilfe der oben erläuterten Klassen einen Baum aufzubauen, der da dann interpretiert werden kann. Ich werde nun die einzelnen Abschnitte dieser Klasse erläutern:
A) Der Parserhamster verwaltet eine Liste, in die alle Variablen gespeichert werden, die er im einzulesenden String findet:

```
package entwurfsmuster.interpreter;

import java.util.List;
import java.util.ArrayList;
```

```
import java.util.StringTokenizer;

public class ParserHamster extends Hamster {

    private List<Variable> variablen;

    public ParserHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
        variablen = new ArrayList<Variable>();
    }

    public List<Variable> getVariablen() {
        return variablen;
    }
```

B) Die Methode `leseTermEin` liest vom Benutzer einen String ein, der in einen mathematischen Term geparst werden soll. Mithilfe der Klasse `StringTokenizer` wird der eingelesene String in so genannte Tokens zerlegt (d.h. in diesem Fall in einzelne Zeichen), die in einer Liste gespeichert werden:

```
    private List<Object> leseTermEin() {
        variablen.clear();
        Kontext kontext = new Kontext();
        String s = liesZeichenkette("Bitte Term eingeben.");
        StringTokenizer tokenizer = new StringTokenizer(s);
        List<Object> list = new ArrayList<Object>();
        while (tokenizer.hasMoreTokens()) {
            list.add(tokenizer.nextToken());
        }
        return list;
    }
```

C) Die Methode `findeVariablen` benutzt die Klassenmethode `isLetter` von `Character`, um alle Buchstaben in unserer Zeichenliste durch Instanzen der Klasse `Variable` zu ersetzen. Damit sowohl Instanzen von `String` als auch von `Variable` aufgenommen werden können, sollen die Elemente der Liste vom Typ `Object` sein:

```
    private List<Object> findeVariablen(List<Object> list) {
        for (Object token : list) {
            if (Character.isLetter(((String)token).toString().charAt(0))) {
                boolean gefunden = false;
                for (Variable variable : this.variablen) {
                    if (variable.getName().equals(token.toString())) {
                        list.set(list.indexOf(token), variable);
                        gefunden = true;
                    }
                }
                if (!gefunden) {
```

```
                Variable var = new Variable(token.toString());
                list.set(list.indexOf(token), var);
                variablen.add(var);
            }
        }
    }
    return list;
}
```

D) Nachdem die Variablen identifiziert sind, müssen wir nun die Additionen und Subtraktionen finden. Da diese immer in einer Dreierform vorliegen – Operand Operator Operand – liest unsere Hilfsmethode `parseTerm` drei Parameter ein. Wir werden später beim Aufruf der Methode sehen, dass dies immer drei aufeinanderfolgende Elemente der schon erwähnten Liste sind.

```
private Term findeTerm(Object obj_1, Object obj_2, Object obj_3) {
    if (obj_1 instanceof Term && obj_3 instanceof Term) {
        if (obj_2.toString().equals("+")) {
            return new AddTerm((Term) obj_1, (Term) obj_3);
        }
        if (obj_2.toString().equals("-")) {
            return new SubTerm((Term) obj_1, (Term) obj_3);
        }
    }
    return null;
}
```

E) Wenn nun mit obiger Methode beispielsweise ein Term

$$(A + B) - (C - D)$$

in

$$(addTerm[A, B]) - (subTerm[C, D])$$

verwandelt wurde, müssen noch die Klammern entfernt werden. Die Methode `parseKlammern` gibt genau dann `true` zurück, wenn von drei aufeinander folgenden Elementen in der Liste das erste Element eine öffnende Klammer, das dritte eine schließende und das zweite ein Term ist.

```
private boolean klammernDa(Object obj_1, Object obj_2, Object obj_3) {
    if (obj_1.toString().equals("(") && obj_3.toString().equals(")")) {
        if (obj_2 instanceof Term) {
            return true;
        }
    }
    return false;
}
```

F) Die Methode `raeumeListeAuf` ist eine Hilfsmethode, die nach dem Parsen eines Terms oder der Entfernung von Klammern, die überflüssigen Elemente aus der Liste entfernt und diese entsprechend verkürzt.

```
    private List<Object> raeumeListeAuf (List<Object> list, int index) {
        for (int j = (index + 1); j < (list.size() - 2); j++) {
            list.set(j, list.get (j + 2));
        }
        list.remove(list.size() - 1);
        list.remove(list.size() - 1);
        return list;
    }
```

G) Die Methode `parseString` schließlich bedient sich der gerade vorgestellten Methoden, um aus einem String einen Term zu parsen.

```
    public Term parseString() {
        List<Object> list = leseTermEin();
        list = findeVariablen(list);
        while (list.size() > 1) {
            //Sub und Add Terme suchen
            for (int i = 0; i < (list.size() - 2); i++) {
                Term t = findeTerm(list.get(i), list.get(i+1),
                                   list.get(i+2));
                if (t != null) {
                    list.set(i, t);
                    list = raeumeListeAuf(list, i);
                }
            }
           //Klammern entfernen
           for (int i = 0; i < (list.size() - 2); i++) {
               if (klammernDa(list.get(i), list.get(i+1), list.get(i+2))) {
                   list.set(i, list.get(i+1));
                   list = raeumeListeAuf(list, i);
               }
           }
       }
       return (Term) list.get(0);
    }
}
```

Der Kontexthamster holt sich vom Parserhamster die Variablen aus dem Term (letzterer hat die sich „nebenbei“ gemerkt) und fragt die Belegung vom Benutzer ab. Diese wird in einer `Kontext`-Instanz gespeichert und zurückgegeben.

```
package entwurfsmuster.interpreter;

import java.util.List;
```

```
public class KontextHamster extends Hamster {

    public KontextHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }

    public Kontext leseKontextEin(List<Variable> variablen) {
        Kontext kontext = new Kontext();
        schreib("Sie haben folgende Variablen benutzt: ");
        for (Variable variable : variablen) {
            int zahl = liesZahl(variable.getName()
                                + ". Geben Sie bitte eine Belegung an.");
            kontext.weiseZu(variable, zahl);
        }
        return kontext;
    }
}
```

Der InterpreterHamster schließlich ruft die `werteAus`-Methode des Terms auf, der den obersten Knoten in der Baumstruktur darstellt. Da dadurch quasi-rekursiv alle weiteren `werteAus`-Methoden aufgerufen werden, geschieht der eigentliche Interpreter-Vorgang „automatisch“.

```
package entwurfsmuster.interpreter;

public class InterpreterHamster extends Hamster {

    public InterpreterHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }

    public void interpretiere(Term term, Kontext kontext) {
        int ergebnis = term.werteAus(kontext);
        schreib("Das Ergebnis lautet: " + ergebnis);
    }
}
```

Im Hauptprogramm werden die drei Hamster erzeugt, danach führen sie in der Reihenfolge ihre Aufgaben durch.
Bei der Ausführung ist zu beachten, dass bei der Eingabe des Terms zwischen den einzelnen Zeichen immer Leerzeichen stehen müssen, da der `StringTokenizer` in dieser Version solche Zeichen als Trennzeichen erwartet. Wird dies nicht beachtet, funktioniert das Programm nicht korrekt!

```
import entwurfsmuster.interpreter.ParserHamster;
import entwurfsmuster.interpreter.KontextHamster;
import entwurfsmuster.interpreter.InterpreterHamster;
import entwurfsmuster.interpreter.Term;
import entwurfsmuster.interpreter.Kontext;
```

```
void main() {
    ParserHamster parserHamster = new ParserHamster(0, 0, 0, 0);
    KontextHamster kontextHamster = new KontextHamster(0, 1, 0, 0);
    InterpreterHamster interpreterHamster =
            new InterpreterHamster(0, 2, 0, 0);

    Term term = parserHamster.parseString();
    Kontext kontext =
            kontextHamster.leseKontextEin(parserHamster.getVariablen());
    interpreterHamster.interpretiere(term, kontext);
}
```

3.8.4 Zusammenfassung und Ausblick

Das Entwurfsmuster Interpreter ist ein klassenbasiertes Verhaltensmuster. Abbildung 12 zeigt ein Klassendiagramm, das abstrakt die wesentlichen Merkmale des Musters zusammenfasst.

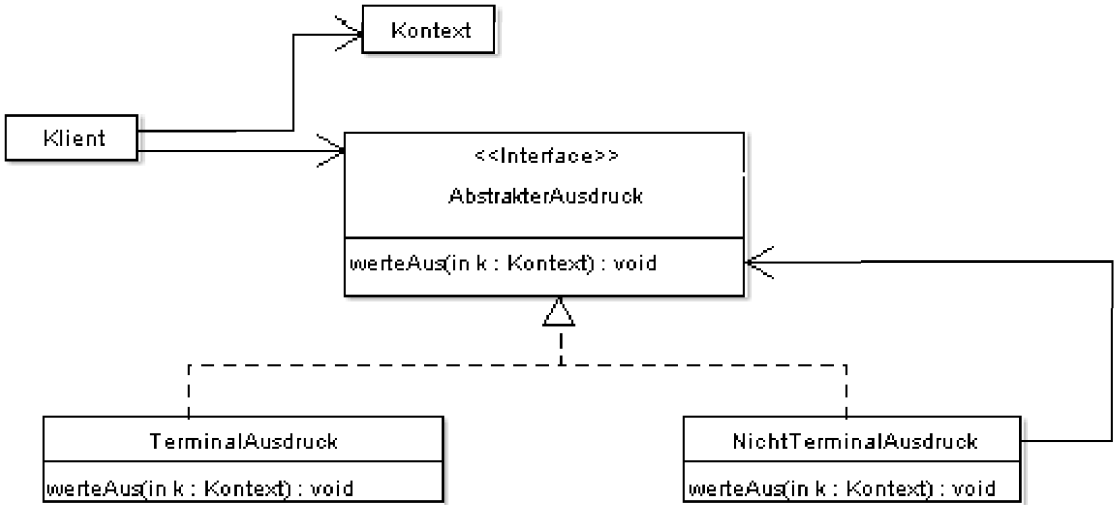


Abbildung 12: Klassendiagramm zum Entwurfsmuster Interpreter

Dieses Entwurfsmuster sollte nur angewandt werden, wenn die Grammatik der zu interpretierenden Sprache einen gewissen Grad an Komplexität nicht überschreitet. Dafür gibt es zwei Gründe:

- Bei einfachen Grammatiken sorgt die Repräsentation von Grammatikregeln durch Klassen für eine größere Übersichtlichkeit. Aber gerade diese Vorgehensweise wird bei komplexen Grammatiken mehr und mehr zum Nachteil, da eine große Klassenstruktur schwer zu verwalten ist.
- Das Interpretermuster ist nicht auf Effizienz optimiert. Bei komplexen Grammatiken stellen Hilfsmittel wie Parsergeneratoren die bessere Alternative dar.

3.9 Strategie (Strategy)

3.9.1 Motivationsaufgabe

Entwerfe mehrere Hamster, die unterschiedliche Algorithmen zum Sammeln von Körnern einsetzen.

3.9.2 Zum Territorium

Benötigt wird ein mittelgroßes Territorium mit darin zufällig verteilten Körnern.

3.9.3 Lösung A: intuitive Vorgehensweise

Für jede Sammelstrategie wird eine eigene Hamsterklasse angelegt, die bis auf eine Methode `sammeln` identisch sind. In dieser wird dann jeweils spezifisch die Sammelstrategie untergebracht.
Die Klasse `SimplerStrategieHamster` benutzt den Sammelalgorithmus aus dem Beobachter-Muster-Beispiel (siehe Kapitel 3.1 auf Seite 22):

```
package entwurfsmuster.strategie.strategieA;

import entwurfsmuster.AllroundHamster;

public class SimplerSammelHamster extends AllroundHamster {

    public SimplerSammelHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }

    public void sammle() {
        int zufall = (int) (Math.random() * 4);
        switch (zufall) {
        case 0:
            vorUndNimm();
            break;
        case 1:
            linksUm();
            vorUndNimm();
            break;
        case 2:
            rechtsUm();
            vorUndNimm();
            break;
        case 3:
            kehrt();
            vorUndNimm();
        }
    }

    public String toString() {
```

```
        return "SimplerSammelHamster hat " + getAnzahlKoerner()
                 + " Körner im Maul.";
    }
}
```

Der `VerbesserteSammelHamster` verbessert die vorangegangene Methode insoweit, als dass er sich merkt, auf welchen Feldern er schon war und versucht, sie zu meiden (weil dort kein Korn mehr liegen kann). Wenn er allerdings keine andere Möglichkeit hat, als ein schon besuchtes Feld zu betreten, dann tut er dies:

```
package entwurfsmuster.strategie.strategieA;

import java.util.List;
import java.util.ArrayList;
import java.util.Arrays;
import entwurfsmuster.AllroundHamster;

public class VerbesserterSammelHamster extends AllroundHamster {

  private List<int[]> pos;

    public VerbesserterSammelHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
        this.pos = new ArrayList<int[]>();
    }

    public void sammle() {
        int[] aktuellePosition = {getReihe(), getSpalte()};
        pos.add(aktuellePosition);
        List<int[]> moeglich = new ArrayList<int[]>();
        int[] richtungN = {getReihe() - 1, getSpalte()};
        int[] richtungO = {getReihe(), getSpalte() + 1};
        int[] richtungS = {getReihe() + 1, getSpalte()};
        int[] richtungW = {getReihe(), getSpalte() - 1};
        boolean nord = true;
        boolean ost = true;
        boolean sued = true;
        boolean west = true;
        for (int[] position : pos) {
            if (Arrays.equals(position, richtungN)) {
                nord = false;
            }
            if (Arrays.equals(position, richtungO)) {
                ost = false;
            }
            if (Arrays.equals(position, richtungS)) {
                sued = false;
            }
            if (Arrays.equals(position, richtungW)) {
```

```
            west = false;
        }
    }
    if (nord) {
        moeglich.add(richtungN);
    }
    if (ost) {
        moeglich.add(richtungO);
    }
    if (sued) {
        moeglich.add(richtungS);
    }
    if (west) {
        moeglich.add(richtungW);
    }
    if (moeglich.size() < 1) {
        moeglich.add(richtungN);
        moeglich.add(richtungO);
        moeglich.add(richtungS);
        moeglich.add(richtungW);
    }
    int zufall = (int) (Math.random() * moeglich.size());
    int[] gewaehlteRichtung = moeglich.get (zufall);
    if (Territorium.mauerDa(gewaehlteRichtung[0],
                            gewaehlteRichtung[1])) {
        pos.add (gewaehlteRichtung);
    }
    if (Arrays.equals(gewaehlteRichtung, richtungN)) {
        setzeBlickrichtung(Hamster.NORD);
        vorUndNimm();
    }
    if (Arrays.equals(gewaehlteRichtung, richtungO)) {
        setzeBlickrichtung(Hamster.OST);
        vorUndNimm();
    }
    if (Arrays.equals(gewaehlteRichtung, richtungS)) {
        setzeBlickrichtung(Hamster.SUED);
        vorUndNimm();
    }
    if (Arrays.equals(gewaehlteRichtung, richtungW)) {
        setzeBlickrichtung(Hamster.WEST);
        vorUndNimm();
    }
}

public String toString() {
    return "VerbesserterSammelHamster hat " + getAnzahlKoerner()
             + " Körner im Maul.";
```

```
    }
}
```

Der `SuperSammelHamster` greift auf Methoden der Klasse `Territorium` zurück, um noch zielgerichteter suchen zu können. Er sucht seine Umgebung ßpiralförmig“ (siehe Abbildung 98) ab, um ein Korn zu finden, das möglichst nah an ihm dran liegt und läuft dann gezielt hin.

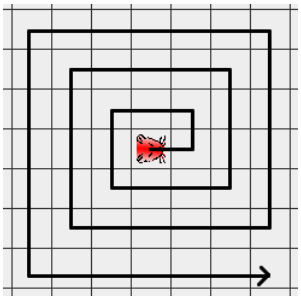


Abbildung 13: Kornsuchmethode des „Supersammelhamsters“

```
package entwurfsmuster.strategie.strategieA;

import entwurfsmuster.AllroundHamster;

public class SuperSammelHamster extends AllroundHamster {

    private int zielReihe;
    private int zielSpalte;

    public SuperSammelHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
        this.zielReihe = -1;
        this.zielSpalte = -1;
    }

    public void sammle() {
        if (getReihe() == this.zielReihe
                && getSpalte() == this.zielSpalte) {
            if (kornDa()) {
                nimm();
            }
            this.zielReihe = -1;
            this.zielSpalte = -1;
        } else if (this.zielReihe == -1 && this.zielSpalte == -1) {
            sucheKorn();
        }
        if (this.zielReihe != getReihe()) {
            if (this.zielReihe > getReihe()) {
                setzeBlickrichtung(Hamster.SUED);
```

```
        } else {
            setzeBlickrichtung(Hamster.NORD);
        }
    } else {
        if (this.zielSpalte > getSpalte()) {
            setzeBlickrichtung(Hamster.OST);
        } else {
            setzeBlickrichtung(Hamster.WEST);
        }
    }
    if (vornFrei()) {
        vor();
    }
}

private int berechneWeg(int reihe, int spalte) {
    int r = Math.abs(getReihe() - reihe);
    int s = Math.abs(getSpalte() - spalte);
    return r + s;
}

private void sucheKorn() {
    int schritte = 1;
    int abstand = 0;
    int blick = getBlickrichtung();
    for (int i = 0; i < 100; i++) {
        for (int j = 0; j < schritte; j++) {
            int reiheAktuell = getReihe();
            int spalteAktuell = getSpalte();
            switch (blick) {
            case Hamster.OST:
                reiheAktuell += abstand;
                spalteAktuell +=j;
                break;
            case Hamster.NORD:
                reiheAktuell -= j;
                spalteAktuell += abstand;
                break;
            case Hamster.WEST:
                reiheAktuell -= abstand;
                spalteAktuell -= j;
                break;
            case Hamster.SUED:
                reiheAktuell += j;
                spalteAktuell -= abstand;
            }
            if (Territorium.getAnzahlKoerner(reiheAktuell,
                                             spalteAktuell) > 0) {
```

```
                    this.zielReihe = reiheAktuell;
                    this.zielSpalte = spalteAktuell;
                    return;
                }
            }
            blick = ((blick + 3) % 4);
            if (i % 2 == 0) {
                schritte++;
            }
            if (i % 4 == 0) {
                abstand++;
            }
        }
    }

    public String toString() {
        return "SuperSammelHamster hat " + getAnzahlKoerner()
                + " Körner im Maul.";
    }
}
```

Im Hauptprogramm wird von jeder Sammelhamsterklasse jeweils eine Instanz erzeugt. Danach darf jeder der Hamster innerhalb einer `while`-Schleife jeweils einen Schritt ausführen. Wir kennen diese Vorgehensweise von dem Hamsterwettrennen (siehe Kapitel 3.2.1 auf Seite 37), um „Gerechtigkeit“ zu garantieren.

```
import entwurfsmuster.strategie.strategieA.SimplerSammelHamster;
import entwurfsmuster.strategie.strategieA.VerbesserterSammelHamster;
import entwurfsmuster.strategie.strategieA.SuperSammelHamster;

void main() {
    SuperSammelHamster s1 = new SuperSammelHamster(1,1,Hamster.OST,0);
    VerbesserterSammelHamster s2 = new VerbesserterSammelHamster(1,1,0,0);
    SimplerSammelHamster s3 = new SimplerSammelHamster(1,1,0,0);
    while (Territorium.getAnzahlKoerner() != 0) {
        s1.sammle();
        s2.sammle();
        s3.sammle();
    }
    Hamster.getStandardHamster().schreib(s1.toString());
    Hamster.getStandardHamster().schreib(s2.toString());
    Hamster.getStandardHamster().schreib(s3.toString());
}
```

3.9.4 Lösung B: erhöhte Wiederverwendbarkeit durch Vererbung

Der größte Nachteil von Lösung A ist, dass alles, was die Hamster gemeinsam haben (und das soll ja alles außer der Sammelstrategie sein) so oft vorhanden ist, wie es Hamster gibt.

Eine Möglichkeit, dieses Problem zu beheben, ist eine Verfahrensweise, die wir schon oft in dieser Arbeit verwendet haben, nämlich die Wiederverwendung von Code durch Vererbung:

1. Man schafft eine abstrakte Klasse `SammelHamster`, die eben jene Gemeinsamkeiten zur Verfügung stellt.
2. Alle konkreten sammelnden Hamster erben von dieser abstrakten Klasse und implementieren nur noch die Methode `sammeln`.

`SammelHamster` würde dann vom Prinzip her so aussehen:

```
package entwurfsmuster.strategie.strategieB;

import entwurfsmuster.AllroundHamster;

abstract public class SammelHamster extends AllroundHamster {

    public SammelHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }

    abstract public void sammle();
}
```

Und die drei konkreten Sammelhamster wie folgt:

```
package entwurfsmuster.strategie.strategieB;

public class SimplerSammelHamster extends SammelHamster {

    public SimplerSammelHamster (int r, int s, int b, int k) {
        super (r, s, b, k);
    }

    public void sammle() {...}
    public String toString() {...}
}

package entwurfsmuster.strategie.strategieB;

import java.util.List;
import java.util.ArrayList;
import java.util.Arrays;

public class VerbesserterSammelHamster extends SammelHamster {

    private List<int[]> pos;

    public VerbesserterSammelHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
```

```
        this.pos = new ArrayList<int[]>();
    }

    public void sammle() {...}
    public String toString() {...}
}

package entwurfsmuster.strategie.strategieB;

public class SuperSammelHamster extends SammelHamster {

    private int zielReihe;
    private int zielSpalte;

    public SuperSammelHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
        this.zielReihe = -1;
        this.zielSpalte = -1;
    }

    public void sammle() {...}
    private int berechneWeg(int reihe, int spalte) {...}
    private void sucheKorn() {...}
    public String toString() {...}
}
```

Ich weise darauf hin, dass in diesem Beispiel die Wiederverwendung von Code gar nicht genutzt werden kann, da der Hamster nichts anderes tut, als Körner zu sammeln. Hätte er jedoch noch andere Methoden, die allen drei Sammelhamstern gemeinsam wären, könnte man diese in ihrer Oberklasse `SammelHamster` implementieren.
Ein weiterer Vorteil der Vererbung ist noch nicht erwähnt worden: Mittels Polymorphie können wir nun die konkreten Objektinstanzen verschiedener Sammelhamsterklassen Objektreferenzen der gemeinsamen Oberklasse zuweisen. Dadurch ist es möglich, die Hamster z.B. in einem Array vom Typ `SammelHamster` zu speichern.

```
import entwurfsmuster.strategie.strategieB.SammelHamster;
import entwurfsmuster.strategie.strategieB.SimplerSammelHamster;
import entwurfsmuster.strategie.strategieB.VerbesserterSammelHamster;
import entwurfsmuster.strategie.strategieB.SuperSammelHamster;

void main() {
  SammelHamster s1 = new SuperSammelHamster (0,0,Hamster.OST,0);
  SammelHamster s2 = new VerbesserterSammelHamster (0,0,0,0);
  SammelHamster s3 = new SimplerSammelHamster (0,0,0,0);
  while (Territorium.getAnzahlKoerner() != 0) {
    s1.sammeln();
    s2.sammeln();
    s3.sammeln();
  }
```

```
  Hamster.getStandardHamster().schreib ("s1 hat " + s1.getAnzahlKoerner()
                                        + " s2 hat " + s2.getAnzahlKoerner()
                                        + " s3 hat " + s3.getAnzahlKoerner()
                                        + ".");
}
```

3.9.5 Lösung C: Optimale Flexibilität durch das Strategie-Muster

Nehmen wir an, wir haben uns auf oben genannte Weise eine Bibliothek von Sammelstrategien geschaffen. Nehmen wir weiterhin an, wir wollen nun in einem Hamster, der bereits von einer anderen Klasse erbt, eine unserer Sammelstrategien benutzen lassen. Dann hätten wir ein weiteres Problem: Wie schon diskutiert, (siehe Kapitel 3.3, Seite 44) gibt es in Java keine Mehrfachvererbung. Doch die Lösung für dieses Problem haben wir mit dem Adapter-Muster gefunden, welches auch hier anwendbar wäre.
In unserem Fall bietet sich etwas an, was konsequenter und präziser erscheint: Wir können die Sammelstrategien direkt in eigenen Klassen kapseln. Damit wenden wir das *Strategie-Muster* an. Ein Hamster, der sammeln können soll, bekommt also einfach ein Attribut `strategie`, an das er die Aufgabe delegiert.

```
package entwurfsmuster.strategie.strategieC;

import entwurfsmuster.AllroundHamster;

public class SammelHamster extends AllroundHamster {

    private Strategie strategie;

    public SammelHamster(int r, int s, int b, int k,
                         Strategie strategie) {
        super(r, s, b, k);
        this.strategie = strategie;
    }

    public Strategie getStrategie() {
        return strategie;
    }

    public void setStrategie(Strategie strategie) {
        this.strategie = strategie;
    }

    public void sammle() {
        this.strategie.sammle(this);
    }
}
```

Um die schnelle Austauschbarkeit von Strategien zu gewährleisten, gibt es ein Interface `Strategie`, welches konkrete Strategien implementieren müssen:

```
package entwurfsmuster.strategie.strategieC;

public interface Strategie {
    public void sammle(SammelHamster hamster);
    public String gebeAus(Hamster hamster);
}
```

Und hier die drei Strategien:

```
package entwurfsmuster.strategie.strategieC;

public class SimpleStrategie implements Strategie {

    public void sammle(SammelHamster hamster) {
        int zufall = (int) (Math.random() * 4);
        switch (zufall) {
        case 0:
            hamster.vorUndNimm();
            break;
        case 1:
            hamster.linksUm();
            hamster.vorUndNimm();
            break;
        case 2:
            hamster.rechtsUm();
            hamster.vorUndNimm();
            break;
        case 3:
            hamster.kehrt();
            hamster.vorUndNimm();
        }
    }

    public String gebeAus(Hamster hamster) {
        return "Hamster mit simpler Strategie hat "
                + hamster.getAnzahlKoerner()
                + " Körner im Maul.";
    }
}
```

```
package entwurfsmuster.strategie.strategieC;

import java.util.List;
import java.util.ArrayList;
import java.util.Arrays;

public class VerbesserteStrategie implements Strategie {

    private List<int[]> pos;
```

```
public VerbesserteStrategie() {
    this.pos = new ArrayList<int[]>();
}

public void sammle(SammelHamster hamster) {
    int[] aktuellePosition = { hamster.getReihe(), hamster.getSpalte() };
    pos.add(aktuellePosition);
    List<int[]> moeglich = new ArrayList<int[]>();
    int[] richtungN = { hamster.getReihe() - 1, hamster.getSpalte() };
    int[] richtungO = { hamster.getReihe(), hamster.getSpalte() + 1 };
    int[] richtungS = { hamster.getReihe() + 1, hamster.getSpalte() };
    int[] richtungW = { hamster.getReihe(), hamster.getSpalte() - 1 };
    boolean nord = true;
    boolean ost = true;
    boolean sued = true;
    boolean west = true;
    for (int[] position : pos) {
        if (Arrays.equals(position, richtungN)) {
            nord = false;
        }
        if (Arrays.equals(position, richtungO)) {
            ost = false;
        }
        if (Arrays.equals(position, richtungS)) {
            sued = false;
        }
        if (Arrays.equals(position, richtungW)) {
            west = false;
        }
    }
    if (nord) {
        moeglich.add(richtungN);
    }
    if (ost) {
        moeglich.add(richtungO);
    }
    if (sued) {
        moeglich.add(richtungS);
    }
    if (west) {
        moeglich.add(richtungW);
    }
    if (moeglich.size() < 1) {
        moeglich.add(richtungN);
        moeglich.add(richtungO);
        moeglich.add(richtungS);
        moeglich.add(richtungW);
```

```
        }
        int zufall = (int) (Math.random() * moeglich.size());
        int[] gewaehlteRichtung = moeglich.get (zufall);
        if (Territorium.mauerDa(gewaehlteRichtung[0],
                                gewaehlteRichtung[1])) {
            pos.add(gewaehlteRichtung);
        }
        if (Arrays.equals(gewaehlteRichtung, richtungN)) {
            hamster.setzeBlickrichtung(Hamster.NORD);
            hamster.vorUndNimm();
        }
        if (Arrays.equals(gewaehlteRichtung, richtungO)) {
            hamster.setzeBlickrichtung(Hamster.OST);
            hamster.vorUndNimm();
        }
        if (Arrays.equals(gewaehlteRichtung, richtungS)) {
            hamster.setzeBlickrichtung(Hamster.SUED);
            hamster.vorUndNimm();
        }
        if (Arrays.equals(gewaehlteRichtung, richtungW)) {
            hamster.setzeBlickrichtung(Hamster.WEST);
            hamster.vorUndNimm();
        }
    }

    public String gebeAus(Hamster hamster) {
        return "Hamster mit verbesserter Strategie hat "
                 + hamster.getAnzahlKoerner()
                 + " Körner im Maul.";
    }
}
```

```
package entwurfsmuster.strategie.strategieC;

import java.util.List;
import java.util.ArrayList;

public class SuperStrategie implements Strategie {

    private int zielReihe;
    private int zielSpalte;

    public SuperStrategie() {
        this.zielReihe = -1;
        this.zielSpalte = -1;
    }

    public void sammle(SammelHamster hamster) {
```

```
    if (hamster.getReihe() == this.zielReihe
           && hamster.getSpalte() == this.zielSpalte) {
        if (hamster.kornDa()) {
            hamster.nimm();
        }
        this.zielReihe = -1;
        this.zielSpalte = -1;
    } else if (this.zielReihe == -1 && this.zielSpalte == -1) {
        sucheKorn(hamster);
    }
    if (this.zielReihe != hamster.getReihe()) {
        if (this.zielReihe > hamster.getReihe()) {
            hamster.setzeBlickrichtung(Hamster.SUED);
        } else {
            hamster.setzeBlickrichtung(Hamster.NORD);
        }
    } else {
        if (this.zielSpalte > hamster.getSpalte()) {
            hamster.setzeBlickrichtung(Hamster.OST);
        } else {
            hamster.setzeBlickrichtung(Hamster.WEST);
        }
    }
    if (hamster.vornFrei()) {
        hamster.vor();
    }
}

private int berechneWeg(Hamster hamster, int reihe, int spalte) {
    int r = Math.abs(hamster.getReihe() - reihe);
    int s = Math.abs(hamster.getSpalte() - spalte);
    return r + s;
}

private void sucheKorn (Hamster hamster) {
    int schritte = 1;
    int abstand = 0;
    int blick = hamster.getBlickrichtung();
    for (int i = 0; i < 100; i++) {
        for (int j = 0; j < schritte; j++) {
            int reiheAktuell = hamster.getReihe();
            int spalteAktuell = hamster.getSpalte();
            switch (blick) {
            case Hamster.OST:
                reiheAktuell += abstand;
                spalteAktuell +=j;
                break;
            case Hamster.NORD:
```

```
                    reiheAktuell -= j;
                    spalteAktuell += abstand;
                    break;
                case Hamster.WEST:
                    reiheAktuell -= abstand;
                    spalteAktuell -= j;
                    break;
                case Hamster.SUED:
                    reiheAktuell += j;
                    spalteAktuell -= abstand;
                }
                if (Territorium.getAnzahlKoerner(reiheAktuell,
                                                 spalteAktuell) > 0) {
                    this.zielReihe = reiheAktuell;
                    this.zielSpalte = spalteAktuell;
                    return;
                }
            }
            blick = ((blick + 3) % 4);
            if (i % 2 == 0) {
                schritte++;
            }
            if (i % 4 == 0) {
                abstand++;
            }
        }
    }

    public String gebeAus(Hamster hamster) {
        return "Hamster mit Super-Strategie hat "
                 + hamster.getAnzahlKoerner()
                 + " Körner im Maul.";
    }
}
```

Schließlich das neue Hauptprogramm:

```
import entwurfsmuster.strategie.strategieC.SammelHamster;
import entwurfsmuster.strategie.strategieC.SimpleStrategie;
import entwurfsmuster.strategie.strategieC.VerbesserteStrategie;
import entwurfsmuster.strategie.strategieC.SuperStrategie;

void main() {
    SammelHamster s1 = new SammelHamster(1,1,Hamster.OST,0,
                                         new SuperStrategie());
    SammelHamster s2 = new SammelHamster(1,1,0,0,new VerbesserteStrategie());
    SammelHamster s3 = new SammelHamster(1,1,0,0,new SimpleStrategie());
    while (Territorium.getAnzahlKoerner() != 0) {
        s1.sammle();
```

```
        s2.sammle();
        s3.sammle();
    }
    Hamster.getStandardHamster().schreib(s1.getStrategie().gebeAus(s1));
    Hamster.getStandardHamster().schreib(s2.getStrategie().gebeAus(s2));
    Hamster.getStandardHamster().schreib(s3.getStrategie().gebeAus(s3));
}
```

3.9.6 Zusammenfassung und Ausblick

Das Entwurfsmuster Strategie ist ein objektbasiertes Verhaltensmuster. Abbildung 14 zeigt ein Klassendiagramm, das abstrakt die wesentlichen Merkmale des Musters zusammenfasst.

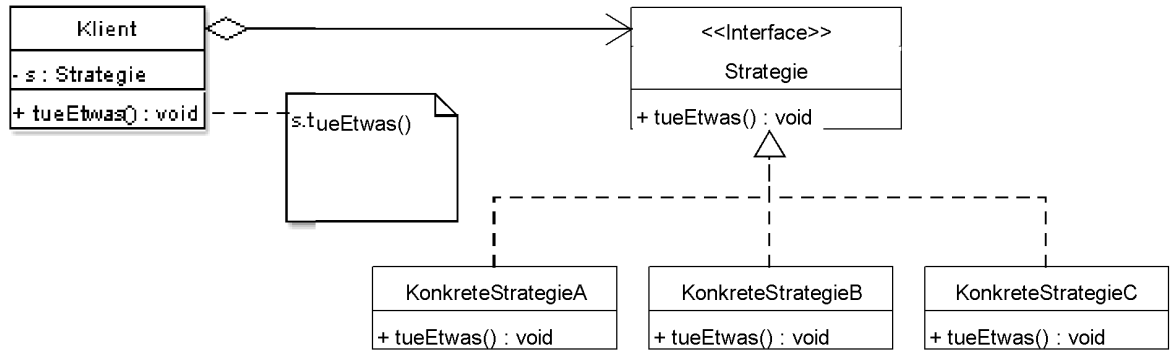


Abbildung 14: Klassendiagramm zum Entwurfsmuster Stragie

In dem vorliegenden Hamsterprogramm haben wir nur Strategie-Klassen auf derselben Ebene in der Klassenhierarchie kennengelernt. Es ist jedoch auch möglich, durch Vererbung Familien von Algorithmen zu entwerfen. Dies bietet sich bei steigender Zahl der Algorithmen an, um die Übersichtlichkeit und die Wiederverwendung des Quellcodes zu erhöhen.

Momentan kann es noch passieren, dass ein `SammelHamster`-Objekt mit `null` als Strategie initialisiert wird. Sobald dann `sammle` aufgerufen wird, wird eine `NullPointerException` geworfen.
Es kann deswegen sinnvoll sein, ein einfaches Default-Verhalten in den Klienten-Klassen zu implementieren für den Fall, dass kein Strategie-Objekt vorhanden ist, an das delegiert werden kann.

3.10 Schablonenmethode (Template Method)

3.10.1 Motivationsaufgabe

Mit dem Strategie-Muster haben wir bereits die Möglichkeit kennengelernt, Algorithmen, die dasselbe Problem lösen, kapseln zu können, um sie austauschbar zu machen.
Nun wollen wir uns mit einem verwandten Thema beschäftigen: Nehmen wir an, wir möchten nicht einen ganzen Algorithmus austauschbar machen, sondern nur Teile davon. Das „Skelett“ des Algorithmus – damit meine ich die Reihenfolge seiner Einzelschritte – soll an einer Stelle fest deklariert sein, während die konkrete Umsetzung der Einzelschritte flexibel gestaltet werden kann. Hat man eine solche Problemstellung, so kann man eine **Schablonenmethode** einsetzen.
Als konkretes Beispiel nehmen wir Hamster, die zu einer bestimmten Kachel hinlaufen, dort ein Korn aufnehmen, sich darüber freuen und schließlich wieder zu ihrer ursprünglichen Kachel zurück laufen. Das sollen alle Hamster gemeinsam haben. Wie sie jedoch die einzelnen Schritte umsetzen, soll bei jedem variiert werden.

3.10.2 Zum Territorium

Es muss mindestens ein Feld geben, auf dem Körner liegen, zu dem die Hamster hinlaufen können. Weiterhin sollte es keine Mauern im Territorium geben, damit es die Hamster leichter haben, hin und her zu laufen.

3.10.3 Die Schablone

Die abstrakte Klasse `KornHolHamster` stellt unsere Schablone dar. In der Methode `holKorn` wird das erwähnte Skelett des Algorithmus implementiert. Um die einzelnen Schritte des Algorithmus flexibel austauschen zu können, werden für sie Methoden aufgerufen, die in dieser Klasse aber nur abstrakt deklariert sind. Eine erbende Klasse kann dann den konkreten Ablauf des Algorithmus mit der Implementierung dieser Methoden flexibel ausgestalten.

```
package entwurfsmuster.schablone;

import entwurfsmuster.AllroundHamster;

abstract public class KornHolHamster extends AllroundHamster {

    protected int startReihe;
    protected int startSpalte;

    public KornHolHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
        this.startReihe = r;
        this.startSpalte = s;
    }

    public void holKorn(int reihe, int spalte) {
        laufHin(reihe, spalte);
        nimm();
        freueDich();
```

```
        laufZurueck();
    }

    abstract public void laufHin(int reihe, int spalte);
    abstract public void freueDich();
    abstract public void laufZurueck();
}
```

3.10.4 Verwendung der Schablone

Die Klasse `SimplerKornHolHamster` erbt von `KornHolHamster` und implementiert die nötigen Methoden minimal:

```
package entwurfsmuster.schablone;

public class SimplerKornHolHamster extends KornHolHamster {

    public SimplerKornHolHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }

    public void laufHin(int reihe, int spalte) {
        gotoKachel(reihe, spalte);
    }

    public void freueDich() {
        kehrt();
        kehrt();
    }

    public void laufZurueck() {
        gotoKachel(this.startReihe, this.startSpalte);
    }
}
```

Die Klasse `AusgabeKornHolHamster` erbt von `SimplerKornHolHamster`. Als Variation erweitert sie die drei Algorithmenschritte um eine Ausgabe:

```
package entwurfsmuster.schablone;

public class AusgabeKornHolHamster extends SimplerKornHolHamster {

    public AusgabeKornHolHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }

    public void laufHin(int reihe, int spalte) {
        schreib("Jetzt laufe ich hin!");
        super.laufHin(reihe,spalte);
```

```
    }

    public void freueDich() {
        schreib("Jetzt freue ich mich!");
        super.freueDich();
    }

    public void laufZurueck() {
        schreib("Jetzt laufe ich zurück!");
        super.laufZurueck();
    }
}
```

Die Klasse `AndererKornHolHamster` erbt wieder von KornHolHamster.
In der Methode `laufHin` wird die Vorgehensweise von `gotoKachel` umgedreht, in dem zuerst zu der entsprechenden Spalte gelaufen wird, dann zu der entsprechenden Zeile. Die Methode `laufZurueck` delegiert das Zurücklaufen an `laufHin`:

```
package entwurfsmuster.schablone;

public class AndererKornHolHamster extends KornHolHamster {

    public AndererKornHolHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }

    public void laufHin(int reihe, int spalte) {
        if (spalte > getSpalte()) {
            setzeBlickrichtung(Hamster.OST);
        } else {
            setzeBlickrichtung(Hamster.WEST);
        }
        while (spalte != getSpalte()) {
            vor();
        }
        if (reihe > getReihe()) {
            setzeBlickrichtung(Hamster.SUED);
        } else {
            setzeBlickrichtung(Hamster.NORD);
        }
        while (reihe != getReihe()) {
            vor();
        }
    }

    public void freueDich() {
        setzeBlickrichtung(SUED);
        vor();
        kehrt();
```

```
        vor();
        setzeBlickrichtung(WEST);
        vor();
        kehrt();
        vor();
    }

    public void laufZurueck() {
        laufHin(this.startReihe, this.startSpalte);
    }
}
```

3.10.5 Das Hauptprogramm

Im Hauptprogramm werden die drei Varianten des Kornholhamsters erzeugt und danach führen sie der Reihe nach vor, was sie jeweils unter „Körner holen“ verstehen:

```
import entwurfsmuster.schablone.KornHolHamster;
import entwurfsmuster.schablone.SimplerKornHolHamster;
import entwurfsmuster.schablone.AusgabeKornHolHamster;
import entwurfsmuster.schablone.AndererKornHolHamster;

void main() {
KornHolHamster hamster1 = new SimplerKornHolHamster(9, 0, 0, 0);
    KornHolHamster hamster2 = new AusgabeKornHolHamster(9, 0, 0, 0);
    KornHolHamster hamster3 = new AndererKornHolHamster(9, 0, 0, 0);

    hamster1.holKoerner(0, 9);
    hamster2.holKoerner(0, 9);
    hamster3.holKoerner(0, 9);
}
```

3.10.6 Zusammenfassung und Ausblick

Das Entwurfsmuster Schablonenmethode ist ein klassenbasiertes Verhaltensmuster. Abbildung 15 zeigt ein Klassendiagramm, das abstrakt die wesentlichen Merkmale des Musters zusammenfasst.
Es erscheint grundsätzlich sinnvoll, beim Entwurf von Schablonenmethoden darauf zu achten, dass die Implementierung von Unterklassen so leicht wie möglich gemacht wird. Dies kann dadurch geschehen,

- dass man die Zahl der in der Oberklasse abstrakt definierten Methoden minimal hält,
- und dass man – wenn möglich – so genannte *Einschubmethoden* anbietet. Einschubmethoden sind in der Oberklasse leer implementiert und werden in der Schablonenmethode dort aufgerufen, wo ein „Einschub“ von beliebigem Code erlaubt wird. Weil sie nicht abstrakt sind, kann eine Unterklasse diese Einschubmethoden ignorieren – oder aber, je nach Bedarf, beliebig überschreiben.

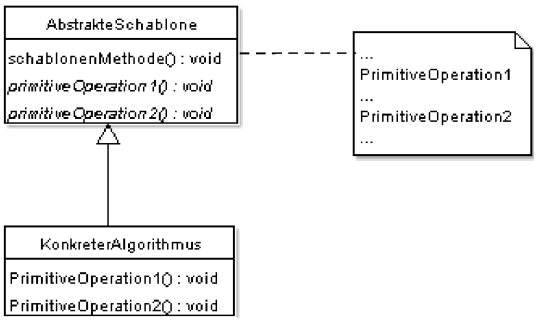


Abbildung 15: Klassendiagramm zum Entwursmuster Schablonenmethode

Außerdem ist darauf hinzuweisen, dass es neben der Unterklassenbildung noch eine andere Möglichkeit gibt, den Algorithmus der Schablonenmethode zu konkretisieren: Durch Delegation an Objekte, die durch Instanzattribute referenziert sind.

3.11 Dekorierer (Decorator)

3.11.1 Motivationsaufgabe

Wir haben bereits kennengelernt, wie man verschiedene Alternativen für eine bestimmte Verhaltensweise des Hamsters kapseln und austauschen kann (siehe Kapitel 3.9 auf Seite 95) und wie man verschiedene Abschnitte eines Algorithmus flexibel gestalten kann (Kapitel 3.10 auf Seite 110)). Wir haben uns aber noch nicht mit dem Thema befasst, wie man das Hamsterverhalten flexibel *erweitern* kann.
Nehmen wir an, wir wollen die Methode `vor` bei verschiedenen Hamstern variieren. Zunächst bietet sich die Unterklassenbildung an, d.h. wir leiten von einer Hamsterklasse ab und überschreiben dann die Methode. Es gibt aber Fälle, in der man eine Unterklassenbildung vermeiden will oder diese gar nicht möglich ist – nämlich dann, wenn die Klasse, von der man ableiten möchte, als `final` deklariert ist.
Für solche Fälle bietet sich das Entwurfsmuster *Dekorierer* an, das ich im Folgenden schrittweise erläutern werde.

3.11.2 Territorium

Es wird ein leeres Territorium ohne Mauern und Körnern benötigt, mit einer ausreichend langen Bahn für die laufenden Hamster.

3.11.3 Die Hamsterkomponente

Wir benötigen zunächst ein Interface `HamsterKomponente`, das alle involvierten Klassen implementieren müssen. Es beinhaltet den Kopf der Methode, die wir variieren wollen:

```
package entwurfsmuster.dekorierer;

public interface HamsterKomponente {
    public void vor();
}
```

3.11.4 Der Laufhamster

Die Klasse `LaufHamster` stellt jene Klasse dar, die die ursprüngliche Methode, die wir variieren wollen, enthält, und die wir nicht ableiten können.

```
package entwurfsmuster.dekorierer;

public final class LaufHamster extends Hamster implements HamsterKomponente {

    public LaufHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }

}
```

3.11.5 Der HamsterDekorierer

Die abstrakte Klasse `HamsterDekorierer` ist das Herzstück des Musters. Sie besitzt ein Instanzattribut vom Typ `HamsterKomponente` und überschreibt die Methode `vor`, indem sie an dieses Attribut delegiert. Abstrakt ist diese Klasse übrigens deswegen, weil es keinen Sinn macht, Objekte von ihr selbst zu erzeugen – sie dient nur als gemeinsame Oberklasse aller konkreten Dekorierer.

```
package entwurfsmuster.dekorierer;

abstract public class HamsterDekorierer implements HamsterKomponente {

    protected HamsterKomponente komponente;

    public HamsterDekorierer(HamsterKomponente komponente) {
        this.komponente = komponente;
    }

    public void vor() {
        komponente.vor();
    }

}
```

3.11.6 Die konkreten Dekorierer

Die Klasse `SchnellDekorierer` erbt nun von `HamsterDekorierer` und überschreibt die Methode `vor`, indem sie dort zwei Mal(!) diese Methode der Oberklasse aufruft.

```
package entwurfsmuster.dekorierer;

public class SchnellDekorierer extends HamsterDekorierer {

    public SchnellDekorierer(HamsterKomponente komponente) {
        super(komponente);
    }

    public void vor() {
        super.vor();
        super.vor();
    }
}
```

Die Klasse `LangsamDekorierer` funktioniert nach demselben Prinzip, allerdings wird hier die Methode `vor` der Oberklasse nur einmal und nur mit einer Wahrscheinlichkeit von fünfzig Prozent (gesteuert durch einen Zufallsgenerator) aufgerufen:

```
package entwurfsmuster.dekorierer;

public class LangsamDekorierer extends HamsterDekorierer {
```

```
    public LangsamDekorierer(HamsterKomponente komponente) {
        super(komponente);
    }

    public void vor() {
        int zufall = (int) (Math.random() * 2);
        if (zufall == 0) {
            super.vor();
        }
    }
}
```

3.11.7 Das Hauptprogramm

Im Hauptprogramm werden nun drei Hamster erzeugt:

1. Ein normaler Laufhamster.
2. Ein dekorierter Laufhamster, der schneller als normal läuft.
3. Ein weiterer dekorierter Laufhamster, der langsamer als normal läuft.

In einer `while`-Schleife wird nun wie üblich die Gleichzeitigkeit für ein Wettrennen simuliert. Zu beachten ist, dass hier jedes Mal die Methode `vor` verwendet wird. Wenn wir das Programm starten, sehen wir, dass die Hamster unterschiedlich schnell laufen, die `vor`-Methode also bei jedem unterschiedlich sein muss. Es ist der Verdienst des Dekorierer-Musters, dass dies möglich ist, obwohl die Methode `vor` der Hamsterklasse LaufHamster nicht überschrieben wurde.
Da wir hier keinen Schiedsrichter wie im Singleton-Beispiel (siehe Kapitel 3.2, Seite 37) zur Verfügung haben, benutzen wir die `MauerDaException`, um den Gewinner dieses Wettrennens zu bestimmen.

```
import entwurfsmuster.dekorierer.HamsterKomponente;
import entwurfsmuster.dekorierer.LaufHamster;
import entwurfsmuster.dekorierer.SchnellDekorierer;
import entwurfsmuster.dekorierer.LangsamDekorierer;

void main() {
    HamsterKomponente hamster = new LaufHamster(0,0,1,0);
    HamsterKomponente schnellerHamster =
            new SchnellDekorierer(new LaufHamster(1,0,1,0));
    HamsterKomponente langsamerHamster =
            new LangsamDekorierer(new LaufHamster(2,0,1,0));
    try {
        while (true) {
            hamster.vor();
            schnellerHamster.vor();
            langsamerHamster.vor();
        }
    } catch (MauerDaException e) {
```

```
        Hamster[] gewinner =
                Territorium.getHamster(e.getReihe(), e.getSpalte() - 1);
        if (gewinner != null && gewinner.length > 1) {
            gewinner[0].schreib("Ich habe gewonnen!");
        }
    }
}
```

3.11.8 Wiederverwendbare Einzelkomponenten

Das Entwurfsmuster Dekorierer hat noch einen weiteren Vorteil, der hier noch nicht angesprochen wurde: Ein Objekt kann nicht nur mit einer „Dekoration“ versehen werden, sondern mit beliebig vielen, die miteinander kombiniert werden können.
Wir haben bereits gesehen, dass ein Dekorierer das Objekt, welches er dekorieren soll, im Konstruktor als Parameter übergeben bekommt. Dieses Objekt muss in unserem Beispiel immer vom Typ `HamsterKomponente sein`. Da ein Dekorierer selbst vom Typ `HamsterKomponente` ist, kann man ihn selbst wieder dekorieren.
Wie dies funktioniert, zeigt das leicht abgeänderte Hauptprogramm:

```
import entwurfsmuster.dekorierer.HamsterKomponente;
import entwurfsmuster.dekorierer.LaufHamster;
import entwurfsmuster.dekorierer.SchnellDekorierer;
import entwurfsmuster.dekorierer.LangsamDekorierer;

void main() {
    HamsterKomponente hamster = new LaufHamster(0,0,1,0);
    HamsterKomponente schnellerHamster =
            new SchnellDekorierer(new LaufHamster(1,0,1,0));
    HamsterKomponente langsamerHamster =
            new LangsamDekorierer(new LaufHamster(2,0,1,0));
    HamsterKomponente weitererHamster = new LangsamDekorierer (
                                            new SchnellDekorierer(
                                                new LaufHamster(3,0,1,0)
                                            )
                                        );
    HamsterKomponente nochEinHamster = new SchnellDekorierer (
                                            new LangsamDekorierer(
                                                new LaufHamster(4,0,1,0)
                                            )
                                       );
    try {
        while (true) {
            hamster.vor();
            schnellerHamster.vor();
            langsamerHamster.vor();
            weitererHamster.vor();
            nochEinHamster.vor();
        }
    } catch (MauerDaException e) {...}
```

```
}
```

Man kann auf diese Art und Weise Dekorierer beliebig schachteln. Schauen wir uns beispielsweise folgende Objekterzeugung an:

```
HamsterKomponente nochEinHamster = new SchnellDekorierer (
                                       new LangsamDekorierer(
                                           new LaufHamster(4,0,1,0)
                                       )
                                   );
```

Wenn nun von dem Objekt `nochEinHamster` die Methode `vor` aufgerufen wird, ruft dies nacheinander die `vor`-Methoden der inneren dekorierten Objekte auf. Dies geschieht mit Hilfe der `super`-Aufrufe und der Delegiertätigkeit der Oberklasse `HamsterDekorierer`. Abbildung 16 zeigt, in welcher Reihenfolge die `vor`-Methoden für dieses Beispiel aufgerufen werden.

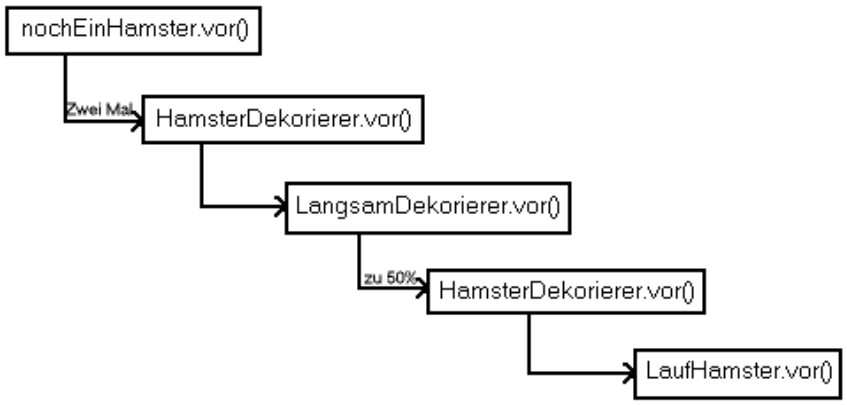


Abbildung 16: Aufrufreihenfolge der vor-Methoden des Objekts nochEinHamster

Zusammengefasst bedeutet dies, dass die Methode `vor` des `LaufHamster` zwei Mal mit fünfzig Prozent Wahrscheinlichkeit aufgerufen wird. Damit ist `nochEinHamster` im statistischen Mittel ungefähr gleich schnell wie der undekorierte `LaufHamster hamster`.
Auch wenn wir Unterklassenbildung zulassen würden, ist hier dem der Dekorierer vorzuziehen, da mit Unterklassenbildung hier fünf Hamsterklassen für fünf verschiedene Hamster notwendig wären, das Dekorierer-Muster jedoch nur mit einer Hamsterklasse und zwei Dekorierern auskommt.

3.11.9 Zusammenfassung und Ausblick

Das Entwurfsmuster Dekorierer ist ein objektbasiertes Strukturmuster. Abbildung 17 zeigt ein Klassendiagramm, das abstrakt die wesentlichen Merkmale des Musters zusammenfasst.
Um das Verständnis für dieses Entwurfsmuster zu vertiefen, scheint es zweckmäßig, die wesentlichen Unterschiede zwischen ihm und dem Strategie-Muster (siehe Kapitel 3.9, Seite 95) zu betrachten:

- Dieses Entwurfsmuster ändert die äußere Hülle eines Objektes, dass selbst nichts über diese Änderungen weiß. Im Gegensatz dazu verändert das Strategie-Muster das Innere eines Objektes und in diesem Fall weiß es selbst um diesen Umstand.

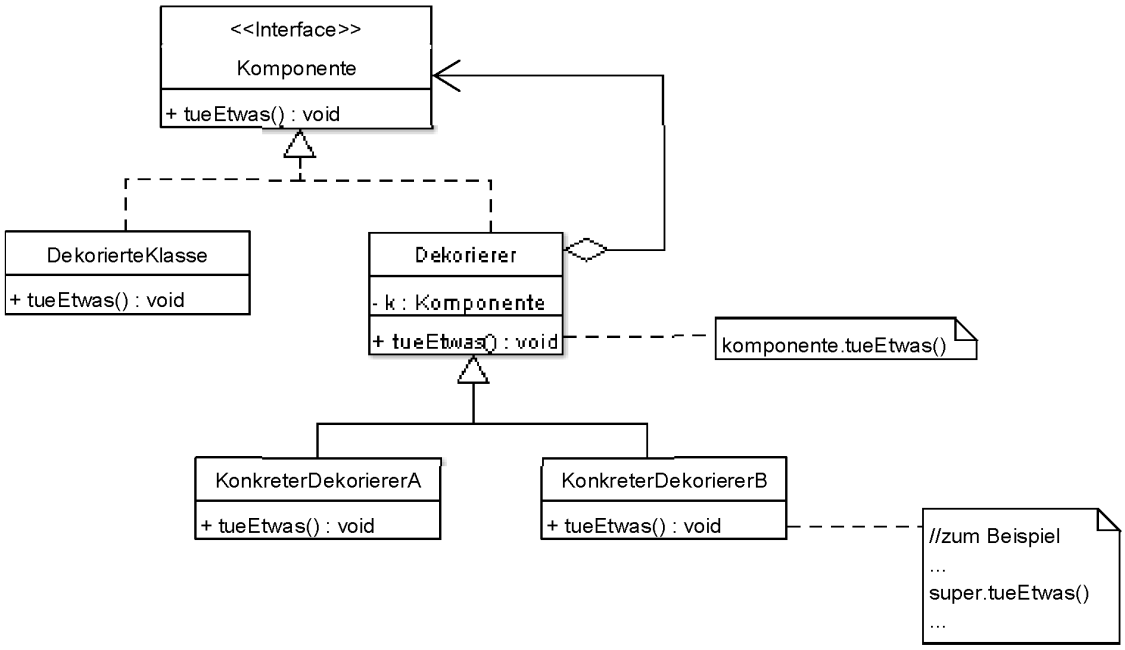


Abbildung 17: Klassendiagramm zum Entwurfsmuster Dekorierer

- Wenn das Strategie-Muster eingesetzt wird, weiß das Objekt, welches die Strategien einsetzt, dass ein Problem vorhanden ist. Beim Dekorierer-Muster muss dies nicht so sein. Das zu dekorierende Objekt könnte zum Beispiel ein Hamster sein, der keine Vorsichtsmaßnahmen dagegen trifft, gegen eine Mauer zu laufen. Dieser Hamster weiß in diesem Fall aber auch gar nichts von Mauern. Eine grundsätzlich andere Vorgehensweise wäre die Implementierung eines Hamsters, dem grundsätzlich die Gefahr der Mauern klar ist, er aber nicht weiß, wie er dieses Problem lösen soll, und es an ein Strategie-Objekt delegiert.

Ein sehr anschauliches Beispiel zum Dekorierer-Muster findet man in [GHJV04] ab S. 206, in der eine Textanzeige mit einem Rahmen und einem Scroll-Balken dekoriert wird.

4 Spiele-Framework

In diesem Kapitel wird die Entwicklung eines Spiele-Frameworks gezeigt und erläutert. Ich schließe damit an das Kapitel 15 („Spielende Hamster“) in [BB04] an und setze die dort geschaffenen Grundlagen zur Spieleprogrammierung beim Leser voraus. Die Bearbeitung der Aufgaben dort wird ebenfalls empfohlen, da ein erhöhtes Verständnis für das Spieleframework zu erwarten ist, wenn man schon einzelne Spiele selbst implementiert hat.

4.1 Anforderungsdefinition

4.1.1 Technische Anforderungen

Framework

Ein Framework (deutsch: „Rahmenwerk“) ist ein parametrisierbares Gerüst von Klassen und Interfaces. Parametrisierbar bedeutet, das es nicht zur Lösung einer speziellen Aufgabe entworfen ist, sondern einen ganzen Aufgabenbereich abdecken soll. Im Framework werden aber nur die Teile implementiert, die alle oder zumindest viele der zu lösenden Aufgaben gemeinsam haben. Um die restlichen speziellen Teile „ankoppeln“ zu können, die eine Aufgabe zusätzlich benötigt, muss das Framework Schnittstellen anbieten.

Der Aufgabenbereich des Frameworks, um das es hier gehen soll, ist die Implementierung von Brettspielen für zwei Personen, bei denen die Gewinnchance nicht durch einen Zufallsfaktor beeinflusst wird.

Parametrisierbar soll dieses Framework in Hinblick auf drei Dinge sein:

1. Es soll zur Implementierung möglichst vieler Spiele geeignet sein. Im optimalen Fall muss nur der Quellcode für ein Spiel geschrieben werden, der die einzigartigen Aspekte dieses Spiels verkörpert.

2. Die Anwendungslogik soll unabhängig von der grafischen Oberfläche sein. Das heißt zum Beispiel, dass in einer Klass `Spielbrett` nicht fest implementiert werden soll, wie das Spielbrett aussieht. Statt dessen soll eine Schnittstelle zur „Beobachtung“ des Spielbretts zur Verfügung stehen, die beliebige Oberflächen (im folgenden als *Views* bezeichnet) das Spielbrett darstellen können.

3. Es sollen folgende Möglichkeiten von Spielerkombinationen vorhanden sein:

   - Mensch gegen Mensch
   - Mensch gegen Computer
   - Computer gegen Computer

   Damit der Computer mitspielen kann, sollen unterschiedliche Spielalgorithmen zur Verfügung gestellt werden bzw. Schnittstellen, die beliebige Spielalgorithmen benutzen können.

Ein Framework soll also ein möglichest abstraktes „Skelett“ bieten, welches von möglichst vielen Programmen genutzt werden kann. Dies wird in Java durch Interfaces realisiert. Um

Programmen, die das Framework nutzen, möglichst viel Freiheit zu lassen, wird also nur vorgeschrieben, *welche* Methoden unbedingt zu implementieren sind, aber nicht *wie*. Außerdem sollen häufig benutzte Programmteile zur Verfügung stehen, damit diese bei der Implementierung von mehreren Spielen wiederverwendet werden können.

Model-View-Controller
Im vorhergehenden Abschnitt haben wir betrachtet, dass das Framework gestaltet sein soll, so dass es zwar im Hamstermodell funktioniert, das Hamstermodell aber nur als ein möglicher View verwendet wird. Andere selbstprogrammierte Views sollen ebenso möglich sein.
Um dies zu realisieren, soll das *Model-View-Controller*-Prinzip verwendet werden. Dies ist ein strukturelles Architekturmuster zur Lösung der Aufgabe, die Benutzerschnittstelle vom funktionalen Kern zu trennen. Im Gegensatz zu Entwurfsmustern – welche lokal eingesetzt werden – betreffen Architekturmuster das ganze System, oder zumindest einen großen Teil davon(siehe [PBG04]).
Das *Model* enthält den funktionalen Kern der Anwendung, der *View* präsentiert dem Anwender die Informationen auf dem Bildschirm und der *Controller* akzeptiert die Bedieneingaben des Anwenders und manipuliert dementsprechend die beiden anderen Komponenten. In Abbildung 18[22] wird verdeutlicht, wie die Trennung von Model und View genutzt werden kann, um dieselben Daten in unterschiedlichen Ansichten zu präsentieren.

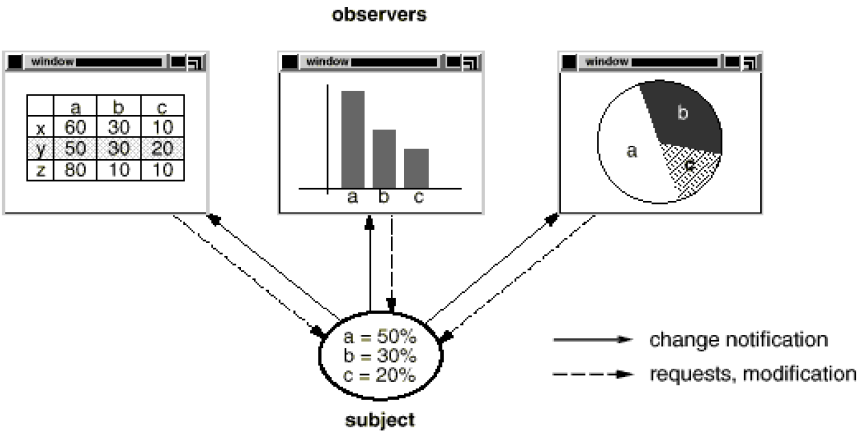


Abbildung 18: Die Model-View-Beziehung

Entwurfsmuster
Es sollen möglichst viele der im Kapitel 3 vorgestellten Entwurfsmuster eingesetzt werden, um deren Zusammenspiel in einem größeren Kontext zu demonstrieren.
Um dieser Anforderung gerecht zu werden, wurden die Muster Beobachter, Abstrakte Fabrik, Strategie, Schablone und Dekorierer eingesetzt. Die genaue Umsetzung der Muster wird an den entsprechenden Stellen in den folgenden Kapiteln und Abschnitten erläutert.

4.1.2 Nicht-technische Anforderungen

In diesem Abschnitt werden die Ergebnisse der Analyse einiger Spiele zusammengetragen, die gemacht wurde, um festzustellen, welche Gemeinsamkeiten sie haben. Auf dieser Basis kann dann im Entwurf die Identifizierung einzelner Klassen und Objekte erfolgen.

[22] entnommen aus [GHJV04]

Folgende Spiele sind betrachtet worden: Schach, Dame, Mühle, Reversi, Alapo, TicTacToe, 4 gewinnt und 5 gewinnt.[23]
Die Anforderungen sind in *soll*- und *kann*-Anforderungen unterteilt. Die *soll*-Anforderungen müssen in jedem Fall erfüllt werden, d.h. Spiele, die diese nicht erfüllen, werden nicht durch das Framework abgedeckt. *kann*-Anforderungen betreffen Funktionalitäten, die einige Spiele betreffen, andere jedoch nicht.

- Ein **Spielbrett** soll mehrere **Spielfelder** haben. Es kann Spielbretter und -felder mit Koordinaten geben.
- Es soll zwei **Spieler** geben.
- Es soll **Figuren** geben. Jede Figur soll zu einem bestimmten Zeitpunkt des Spielbalaufs entweder dem einen oder dem anderen Spieler zugeordnet sein.
- Figuren können von gleicher oder verschiedener Art sein; verschieden in dem Sinne, dass für sie verschiedene Regeln in Bezug auf mögliche Züge gelten und dass sie unterschiedlich dargestellt werden.
- Der Spielablauf soll aus einer Abfolge von **Spielzügen** bestehen, die von einem **Schiedsrichter** überwachtet werden, der die Spielregeln verwaltet.
- Der Schiedsrichter soll nach einem Spielzug entscheiden, welcher Spieler als nächstes an der Reihe ist. Dieser soll dann seinen Spielzug bekanntgeben.
- Die Reihenfolge der Spieler kann immer abwechselnd erfolgen, sie kann aber auch einer anderen Regel folgen.
- Figuren sollen Spielfeldern zugeordnet sein. Es kann eine Figur genau einem Spielfeld zugeordnet sein, es können mehrere Figuren dem gleichen Spielfeld zugeordnet sein, mehreren Spielfeldern kann dieselbe Figur zugeordnet sein.
- Es kann eine initiale Startposition von Figuren auf dem Spielbrett geben.

Diese Aufstellung ist nur ein Ausschnitt aus den Anforderungen, die tatsächlich aufzuzählen möglich wären. Sie soll uns an dieser Stelle aber genügen, da sie ausreichend richtungsweisend ist, um mit dem Entwurf beginnen zu können.
Es sei weiterhin angemerkt, dass die Anforderungsdefinition eines Frameworks in den meisten Fällen nicht zu hundert Prozent abgeschlossen werden, gerade weil sie einen offenen Charakter haben sollen.

4.2 Entwurf

Zu Beginn des Entwurfs müssen die wichtigsten Objekte im System anhand der Anforderungsdefinition identifiziert werden. Um die statischen Elemente eines Brettspiels abzubilden, benötigen wir **Spieler**, ein **Spielbrett**, **Spielfelder** und **Figuren**. Das Grundelement des Spielablaufs ist der **Spielzug**. Die Kontrolle über die Spielregeln hat ein **Schiedrichter** – dieser ist damit zugleich wichtigste Schnittstelle für das Framework benutzende Spiele. Die wichtigste Schnittstelle zur grafischen Oberfläche ist die zur Beobachtung des Spielbretts; wir nennen sie **SpielbrettView**. Schließlich ist der Spielbablauf eingebettet in einem **Spiel**.

[23] Die Spielregeln können z.B. unter www.wikipedia.de eingesehen werden.

Es erscheint sinnvoll, als nächstes zu überlegen, welche Objekte welche anderen kennen sollen, d.h. über Instanzattribute Zugriff auf sie haben sollen. Zum Beispiel erscheint es logisch, dass ein Spielbrett seine Spielfelder kennt – aber auch umgekehrt sollte ein Spielfeld wissen, zu welchem Spielbrett es gehört. In Abbildung 19 ist skizziert, welche Struktur ich diesbezüglich für das Framework verwendet habe.

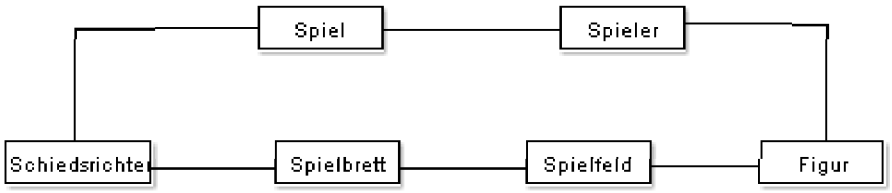


Abbildung 19: Assoziationen zwischen den Basisobjekten im Framework

Übrigens bilden die Spielzüge bilden an dieser Stelle eine Ausnahme und zwar wegen ihrer kurzen Lebensdauer. Ein `Spielzug`-Objekt wird dann erzeugt, wenn ein Spieler seinen Zug bekannt gibt und verfällt, wenn dieser Zug abgeschlossen ist.

In den folgenden Abschnitten werden wir Interfaces für diese *Basiselemente* entwerfen, die festlegen, welche Methoden die entsprechenden Klassen mindestens implementieren müssen.

4.2.1 Spieler

Die wichtigste Methode, die eine `Spieler`-Klasse implementieren muss, ist `getZug`. Weiterhin müssen wir einen Spieler eindeutig identifizieren können. Da es um Spiele für genau zwei Personen gehen soll, bietet sich ein Instanzattribut vom Typ `boolean` an, welches angibt, ob der betreffende Spieler Spieler A oder Spieler B ist. Wenn wir hiervon ausgehen, können wir für die eindeutige Identifizierung eine Methode `istSpielerA` vorschreiben, die einen entsprechenden `boolean`-Wert zurückgibt.
Aus Abbildung 19 können wir entnehmen, dass ein `Spieler` sein `Spiel` und seine `Figur`-Objekte kennen soll. Also fügen wir für beides `get`- und `set`-Methoden hinzu.
Das Interface `Spieler` sieht dann wie folgt aus:

```
package framework;

public interface Spieler {
    public Spielzug getZug();
    public boolean istSpielerA();
    public Spiel getSpiel();
    public void setSpiel(Spiel spiel);
    public Figur[] getFiguren();
    public void setFiguren(Figur[] figuren);
}
```

4.2.2 Spielbrett

Ein Spielbrett besteht aus Spielfeldern, auf denen Figuren stehen. An dieser Stelle müssen wir eine Entwurfsentscheidung treffen: Welches Objekt soll die Kontrolle über die Bewegung

einer Figur haben? In Frage kommen zum Beispiel

- die Figur, die gerade bewegt werden soll,
- das Spielfeld, auf der diese Figur steht,
- und das Spielbrett, zu dem dieses Spielfeld gehört.

Wir haben aber schon gesehen, dass es Spiele gibt, bei denen ein Spielzug darin besteht, eine *neue* Figur auf ein Spielfeld zu setzen. In diesem Fall wäre es kompliziert, die Bewegungskontrolle der Figur selbst zu übertragen, da das entsprechende Objekt erst im „luftleeren“ Raum außerhalb des Brettes erzeugt werden muss, um sich dann selbst den Bestimmungsort zuzuweisen. Aus demselben Grund ist es unmöglich, hier die Kontrolle dem Feld zu geben, auf dem die Figur gerade steht, da sie ja vor der Ausführung des Spielzugs noch gar nicht existiert und somit auch nicht auf einem Feld stehen kann. Als Hilfskonstruktion bliebe dann nur die Möglichkeit, Spielfelder außerhalb des Spielbretts bereitzuhalten, von denen aus Figuren ins Spiel starten können. Es ist gar nicht notwendig, die Implementierung genau vor Augen zu haben, um einzusehen, dass eine solche Lösung nicht gerade intuitiv ist.
Wir entscheiden uns also dafür, dass Klassen die das Spielbrett repräsentieren, die Kontrolle über die Bewegung der Figuren haben sollen, weil hier nicht wie bei den anderen Möglichkeiten auf den ersten Blick erhebliche Nachteile zu erkennen sind. Natürlich wissen wir an dieser Stelle im Entwurfsprozess nicht genau, ob dies die beste Lösung ist – aber wir haben dokumentiert, wie wir zu dieser Entscheidung gekommen sind und welche Alternativen aus welchem Grund ausscheiden. Im schlimmsten Fall könnten wir also später an diesem Punkt ansetzen und auf dieses Erkenntnissen aufbauend eine tiefergehende Analyse machen, wie die Lösung umstrukturiert werden kann.
Wenn wir uns noch einmal die Anforderungsdefinition in Bezug auf Figurenbewegungen ansehen, stellen wir fest, dass sie sich alle aus zwei Elementaroperationen zusammensetzen lassen:

1. Eine Figur wird auf ein Spielfeld gestellt.
2. Eine Figur wird von einem Spielfeld entfernt.

So kann z.B. ein häufig vorkommender Zug, wie das Schlagen einer Figur B durch eine andere Figur A auf dem Brett wie folgt in diese Elementaroperationen zerlegt werden:

- Figur A wird von dem Spielfeld entfernt, auf dem sie gerade steht.
- Figur B wird von dem Spielfeld entfernt, auf dem sie gerade steht.
- Figur A wird auf das Spielfeld gestellt, auf dem Figur B vorher stand.

Wir brauchen also nicht für jeden nur erdenklichen Spielzug extra Methoden bereitstellen, sondern es genügt die Methode `weiseZu`, mit der einem Spielfeld entweder eine Figur zugewiesen oder mit dem Parameter `null` eine Figur entfernt werden kann. Eine häufige Operation, wie das Umwandeln einer Figur in eine andere (z.B. im Schach, wenn ein Bauer die letzte Reihe erreicht) stellen wir mit `wandleUm` ebenfalls zur Verfügung.
Weiterhin gibt es in vielen Spielen eine vorgeschriebene Startaufstellung der Figuren, die wir durch die Methode `initialisiere` unterstützen wollen. Mit der Methode `gehoertDazu` kann man absichern, ob ein angegebenes `Spielfeld` zu einer `Spielbrett`-Instanz gehört. Über die Methode `meldeAn` kann die grafische Oberfläche sich bei dem Spielbrett dafür anmelden, bei

Veränderungen auf dem Spielbrett benachrichtigt zu werden. Wie dies genau geschehen kann, wollen werden wir uns im nächsten Kapitel (Prototyp-Entwicklung) ansehen.
Schließlich entnehmen wir Abbildung 19, dass das `Spielbrett` seinen `Schiedsrichter` und seine `Spielfeld`-Objekte kennen soll.
Somit sieht unser Interface `Spielbrett` bisher so aus:

```
package framework;

public interface Spielbrett {
    public void weiseZu(Spielfeld feld, Figur figur);
    public void wandleUm(Figur alteFigur, Figur neueFigur);
    public void initialisiere();
    public boolean gehoertDazu(Spielfeld feld);
    public void setRichter(Schiedsrichter richter);
    public Schiedsrichter getRichter();
    public void setFelder(Spielfeld[] felder);
    public Spielfeld[] getFelder();
    public void meldeAn(SpielbrettView view);
}
```

4.2.3 Spielfelder

Für die Konzeption des Interface `Spielfeld` müssen wir uns diesmal nur an der Abbildung 19 orientieren, allerdings sollten wir auch einen Blick bezüglich der Spielfelder auf die Anforderungsdefinition werfen: Ein `Spielfeld` kennt das `Spielbrett`, von dem es Teil ist und es kennt die Figur oder die Figuren(!), die auf ihm stehen.
Eine Spielfeldklasse, die diese Anforderung erfüllt, kann seine `Figur`-Objekte in einer Liste verwalten und sollte folgende Methoden implementieren, die zur Modifikation einer solchen Liste sinnvoll sind:

- `entferneFigur`: Bei genau einer Figur pro Spielfeld könnte man für diese Funktionalität auch die `setFigur`-Methode mit dem Parameter `null` verwenden. Bei einer Liste von Figuren scheint es hingegen zweckmäßig zu sein, eine einzelne Figur aus dieser Liste löschen zu können.
- `getFiguren`: Liefert alle Figuren, die auf dem Spielfeld stehen.
- `setFiguren`: Die Figurenliste des Spielfelds wird durch die als Parameter angegebene ersetzt.
- `addFiguren`: Die Figuren der als Parameter angegebenen Liste werden der Figurenliste des Spielfelds *hinzugefügt*.
- `entferneFiguren`: Die Figuren der als Parameter angegebenen Liste werden aus der Figurenliste des Spielfelds gelöscht.

Es wäre jedoch von Nachteil, diese Methoden mit dem Interface `Spielfeld` für alle Spielfelder vorzuschreiben. Denn viele Spiele lassen nur genau eine Figur pro Spielfeld zu; für diese Spiele wären die beiden Methoden `getFigur` und `setFigur` ausreichend und vor allem sinnvoller.
In einer solchen Situation muss man abwägen, ob `ein` Interface, dass alle Anforderungen abdeckt, nicht zu überladen ist und die Implementierung konkreter Spiele unnötig erschwert.

Im vorhergehenden Abschnitt haben wir zum Beispiel betrachtet, dass es Spiele gibt, bei denen es eine Startaufstellung von Figuren gibt und Spiele, die mit einem leeren Spielbrett beginnen. Für die Spiele `mit` Startaufstellung sehen wir die Methode `initialisiere` vor, das ist aber nur ein minimaler Mehrwaufwand bezüglich der Implementierung von Spielen `ohne` Startaufstellung. Diese können die Methode einfach leer implementieren.
Kehren wir nun zum Interface `Spielfeld` zurück. Hier sieht die Situation anders aus als beim `Spielbrett`: Es wäre aufwändig, sinnvolle Lösungen für `addFiguren` oder `setFiguren` für all jene Spiele finden zu müssen, die nur eine Figur pro Spielfeld vorsehen.
Wir können dieses Problem wie folgt lösen:

- Wir definieren ein Interface `Spielfeld` mit den Methoden `getFigur` und `setFigur`. Dies ist die oberste Schnittstelle für Spielfelder und optimal für Spiele mit immer nur einer Figur pro Feld.

- Von `Spielfeld` leiten wir ein Interface `KomplexesSpielfeld` ab, welches die oben genannten Methoden vorschreibt, so dass Klassen, die dieses Interface implementieren, mehrere Figuren pro Spielfeld verwalten können.

Dies hat zwar den Nachteil, dass man wiederum bei Spielen, die `KomplexesSpielfeld` verwenden, einen Mehraufwand hat, weil auch diese `getFigur` und `setFigur` implementieren müssen, hier lassen sich aber einfache Lösungen finden, wie wir im Implementierungsabschnitt sehen werden.
Schließlich dürfen wir die `set`- und die `get`-Methode für das `Spielbrett` nicht vergessen:

```
package framework;

public interface Spielfeld {
    public Figur getFigur();
    public void setFigur(Figur figur);
    public Spielbrett getBrett();
    public void setBrett(Spielbrett brett);
}

package framework;

import java.util.List;

public interface KomplexesSpielfeld extends Spielfeld{
    public void entferneFigur(Figur figur);
    public List<Figur> getFiguren();
    public void setFiguren(List<Figur> figuren);
    public void addFiguren(List<Figur> figuren);
    public void entferneFiguren(List<Figur> figuren);
}
```

4.2.4 Figuren

Ein Blick auf Abbildung 19 verrät uns, dass eine Figur den anderen Basiselementen `Spieler` und `Spielfeld` assoziiert ist. Genauso, wie auf einem Spielfeld mehrere Figuren stehen

können, wollen wir auch – laut Anforderungsdefinition – zulassen, dass eine Figur auf mehreren Spielfeldern stehen kann. Da wir diese Problemstellung auf dieselbe Weise lösen, wie im vorhergehenden Abschnitt, wird sie hier nicht weiter diskutiert.

```
package framework;

public interface Figur {
    public void setSpieler(Spieler spieler);
    public Spieler getSpieler();
    public void setFeld(Spielfeld feld);
    public Spielfeld getFeld();
}
```

```
package framework;

import java.util.List;

public interface KomplexeFigur extends Figur{
    public void entferneFeld(Spielfeld feld);
    public List<Spielfeld> getFelder();
    public void setFelder(List<Spielfeld> felder);
    public void addFelder(List<Spielfeld> felder);
    public void entferneFelder(List<Spielfeld> felder);
}
```

4.2.5 Spielzug

Aus den Überlegungen zum Spielbrett können wir schließen, dass ein Spielzug über die Spielfelder beschrieben werden kann, die von ihm betroffen sind. Es lassen sich zwei häufig vorkommende Grundtypen von Spielzügen feststellen:

- Eine Figur wird von einem Spielfeld zu einem anderen bewegt. Beispiel: Beim Schach wird ein Bauer um ein Feld nach vorn bewegt. Durch Angabe dieser beiden Felder hat man alle wesentlichen Informationen über den Zug, da man über das Startfeld auf die Figur schließen kann, die gezogen wird.
- Eine neue Figur wird auf ein Spielfeld gesetzt. Beispiel: Ein Zug beim TicTacToe-Spiel. Hier genügt die Angabe des Zielfeldes. Anmerkung: Bei allen betrachteten Spiele, bei denen so ein Zug möglich ist, gibt es nur eine Figurenart. Wäre das nicht der Fall, müsste man diese zusätzlich angeben.

Weiterhin kann es Züge geben, die mehr als zwei Spielfelder betreffen: zum Beispiel beim Damespiel, wenn mehrere Sprungzüge nacheinander möglich sind.
Das Interface `Spielzug` muss so beschaffen sein, dass es auf diese drei Spielzugarten passt. Wir gehen davon aus, dass ein `Spielzug`-Objekt mit den entsprechenden `Spielfeld`-Objekten im Konstruktor initialisiert wird, deswegen brauchen wir hier nur darauf einzugehen, wie wir an die Information von außen wieder herankommen.
Um den Anforderungen des zuerst genannten Spielzugtyps gerecht zu werden, schreiben wir die Methoden `getStart` und `getZiel` vor. Für den zweiten Typ reicht `getZiel` aus. Und für den Fall, dass ein Zug mehr als zwei Spielfelder hat, sollte eine Methode `getFelder` zur Verfügung stehen:

```
package framework;

public interface Spielzug {
    public Spielfeld getStart();
    public Spielfeld getZiel();
    public Spielfeld[] getFelder();
}
```

Es lässt sich nicht vermeiden, dass alle Spielzugtypen alle Methoden implementieren müssen, auch wenn manche Kombinationen nicht recht passen. Wie dieses Problem akzeptabel gelöst werden kann, beschäftigt uns aber erst bei der Implementierung der konkreten Klassen in Kapitel 4.4.2, Seite 153.

Langsam nimmt die Struktur des inneren Kerns unseres Frameworks Gestalt an: Wir haben Interfaces für das Spielbrett, seine Spielfelder, die Figuren und die Spieler, die die Figuren bewegen. Nun wird es Zeit, uns um die beiden wesentlichen Schnittstellen nach außen zu kümmern, also um den `Schiedsrichter` und den `SpielbrettView`.

4.2.6 Schiedsrichter

Das Wesentliche, was ein konkretes Spiel einbringen muss – was es von anderen Spielen unterscheidet – sind die Spielregeln. Da das Objekt, was zum Beispiel dafür zuständig ist, zu überprüfen, ob ein Zug korrekt ausgeführt wurde, aktiv und steuernd in den Spielverlauf eingreift, nennen wir es `Schiedsrichter`.
In einer Klasse `Schiedsrichter` sind im wesentlichen fünf Methoden zur Spielregelverwaltung zu vermerken:

- `ermittleSpieler`: Es wird der Spieler ermittelt, der den nächsten Zug machen muss.
- `zugKorrekt`: Diese Methode liefert `true` zurück, wenn der Zug den Spielregeln entspricht, andernfalls `false`.
- `fuehreZugAus`: Führt den angegeben Zug auf dem mit dem Schiedsrichter assozierten Brett aus.
- `spielBeendet`: Diese Methode liefert `true` zurück, falls eine Spielabbruchbedingung wahr ist, z.B. wenn ein Spieler gewonnen hat.
- `verkuendeSieger`: Gibt bei Spielende bekannt, welcher Spieler gewonnen hat.

Weiterhin soll die Methode `getAlleZuege` ein Array mit allen Spielzügen liefern, die in der aktuellen Spielsituation möglich sind. Sie kann intern von `zugKorrekt` verwendet werden, um zu ermitteln, ob ein Zug korrekt ist oder nicht, aber auch extern zum Beispiel für Spielalgorithmen zur Verfügung stehen.

Ein Thema, dass noch nicht besprochen wurde, ist die Umwandlung von Strings, die von menschlichen Spielern eingegeben werden, in äquivalente Spielzüge. Wie überhaupt mit dem Benutzer kommuniziert werden kann, wird in Abschnitt 4.2.8, Seite 131 diskutiert. An dieser Stelle soll es nur um die schon angesprochene String-Spielzug-Umwandlung gehen.

Da Spielzüge durch Spielfelder beschrieben werden können, macht es Sinn, dass Strings für Spielzüge aus einer Aneinanderreihung von Spielfeld-String-Repräsentationen bestehen, mit einem beliebigen Trennzeichen dazwischen. Wenn wir wieder das Schachspiel als Beispiel betrachten, macht dies durchaus Sinn, da die gängige Schachnotation ebenfalls so aufgebaut ist.
Zusätzlich müssen wir einplanen, dass es in manchen Spielen zum Beispiel erlaubt ist, aufzugeben, zu passen oder Remis anzubieten. Dies sind eine Reihe von speziellen Spielzügen, für die wir spezielle Strings vorsehen müssen.
Durch diese Überlegungen sehen wir, dass die String-Spielzug-Umwandlung regelabhängig sein kann, und damit zum Schiedsrichter gehört. Andererseits ist es sinnvoll, diesen Bereich in einem eigenen Objekt zu kapseln, um die Wiederverwendbarkeit zu erhöhen. Wie dies genau geschehen kann, werden wir in Kapitel 4.3 (ab Seite 133) betrachten. An dieser Stelle ist es ausreichend ein passendes Interface zu definieren...

```
package framework;

public interface Interpreter {
    public Spielzug getZug(String str);
}
```

...und eine Methodenvorschrift `getInterpreter` in das Interface `Schiedsrichter` einzufügen. Aus Abbildung 19 entnehmen wir schließlich, das ein `Schiedsrichter`-Objekt ein zugehöriges `Spielbrett` und sein `Spiel` kennt. Wir fügen entsprechende `get`- und `set`-Methoden hinzu und unser Interface ist damit vollständig.

```
package framework;

public interface Schiedsrichter {
    public Spieler ermittleSpieler(Spieler vorherigerSpieler);
    public boolean zugKorrekt(Spielzug zug, Spieler spieler);
    public void fuehreZugAus(Spielzug zug, Spieler spieler);
    public boolean spielBeendet();
    public void verkuendeSieger();

    public Spielbrett getBrett();
    public void setBrett(Spielbrett brett);
    public Spiel getSpiel();
    public void setSpiel(Spiel spiel);

    public Interpreter getInterpreter();
    public Spielzug[] getAlleZuege(Spieler spieler);

    public Schiedsrichter kopiere();

}
```

4.2.7 SpielbrettView

Wir stellen ein Interface `SpielbrettView` bereit, das Views implementieren müssen, die unser Spielbrett abbilden sollen:

```
package framework;

public interface SpielbrettView {
    public void aktualisiere(Spielbrett brett);
}
```

Damit setzen wir eine weitere Vorgabe um: Wir wollen das Architekturmuster Model-View-Controller – und darin eingebettet das Entwurfsmuster Beobachter – verwenden. Dieses Interface stellt die eine Hälfte des Beobachtermusters dar: Es entspricht dem Interface `Empfaenger`, dass wir im Kapitel über das Beobachtermuster kennengelernt haben (Kapitel 3.1.7, Seite 31). Die Weitere Umsetzung des Model-View-Controller wird in Kapitel 4.3 diskutiert.

4.2.8 Zwischenstand des Entwurfs

Das Klassendiagramm in Abbildung 20 zeigt den aktuellen Stand unseres Entwurfs: Das Spielbrett hat etwas mit dem Spielfeld zu tun, dieses mit den Figuren – und diese wiederum mit den Spielern. Außerdem ist der Spielbrett-View mit dem Spielbrett assoziiert. Ansonsten fehlt uns aber noch die entscheidende Verknüpfung der restlichen Klassen.

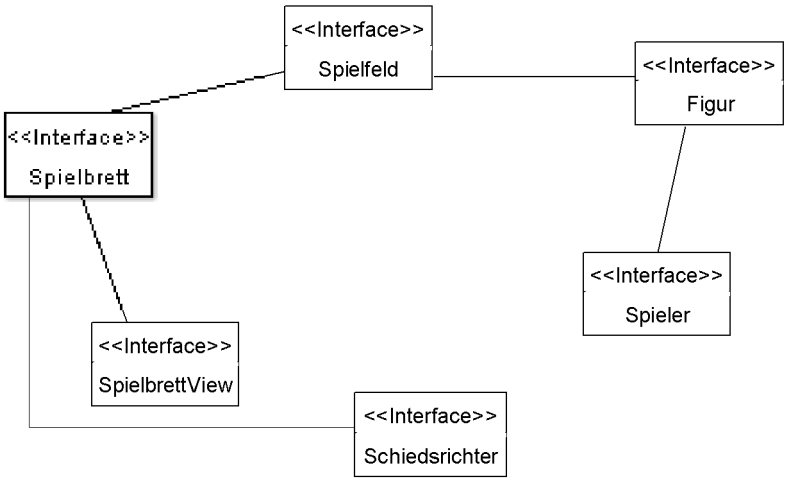


Abbildung 20: Klassendiagramm der bisherigen Basiselemente

Schauen wir uns an, was wir bisher modelliert haben:

- Statische Abbildungen der Elemente, die in der realen Welt der 2-Personen-Brettspiele vorkommen
- Elementare Operationen auf diesen Elementen

Ein Spiel besteht jedoch aus mehr, es gibt nämlich immer auch eine dynamische Komponente, den Spielablauf, der all dies in einen Zusammenhang stellt. Nehmen wir als Beispiel wieder das Schachspiel:

- Es wird gelost, wer die weißen Figuren bekommt, d.h. wer anfängt.
- Das Brett wird aufgestellt, die Figuren werden in die Startaufstellung gebracht.

- Der Spieler mit Weiß fängt an und macht einen Zug, der den Regeln entsprechen muss. Danach spielen die Spieler immer abwechselnd. Passen ist nicht erlaubt.
- Die Spielabbruchbedingung ist das Schachmatt, die Aufgabe eines Spielers oder Unentschieden, z.B. durch Patt oder Remis.

Verallgemeinern wir dies auf den allgemeinen Ablauf von 2-Personen-Brettspielen – wir benutzen dabei die Elemente, die wir modelliert haben:

1. Das Spielbrett und die Startaufstellung (wenn es denn eine gibt) werden in Position gebracht.
2. Der Schiedsrichter entscheidet, welcher Spieler am Zug ist.
3. Jener Spieler sagt seinen Zug, der Schiedsrichter überprüft, ob dieser korrekt ist und führt ihn auf dem Spielbrett aus.
4. Die Schritte 2 und 3 werden solange wiederholt, bis eine Spielabbruchbedingung auftritt (z.b. gibt ein Spieler auf, ein Spieler gewinnt, die Spieler einigen sich auf Unentschieden usw.).
5. Der Gewinner des Spiels wird bekannt gegeben.

Es wirkt zunächst irritierend, dass der Schiedsrichter den Zug ausführt. Hier scheint die reale Welt nicht korrekt abgebildet zu sein. Stellen wir uns jedoch vor, wir spielen gegen einen Computer Schach, dann teilen wir auch zunächst nur unseren Zug per Eingabe mit und eine Instanz innerhalb des Programms überprüft die Korrektheit des Zuges, führt ihn aus, so dass wir letztendlich die Veränderung des Spielbretts auf dem Bildschirm sehen. Selbst zwei menschliche Spieler haben so eine Schiedsrichterinstanz – sie teilen sie sich in der Form, dass immer der Spieler, der nicht an der Reihe ist, kontrolliert, ob der Zug des anderen korrekt ist und ihn ggf. korrigiert.

4.2.9 Das Interface Spiel

Die Konsequenz der vorherigen Überlegungen ist der Entwurf einer Klasse, die sowohl die einzelnen bisherig entworfenen Elemente miteinander verknüpft als auch den Spielablauf in einer Methode zur Verfügung stellt (siehe dazu Abbildung 21).

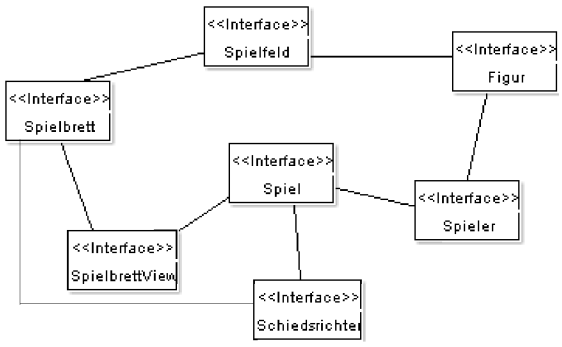


Abbildung 21: Klassendiagramm mit einer Klasse Spiel

Wir wollen uns hier mit dem Aufbau des Interface beschäftigen, welches vorschreibt, welche Methoden eine solche Klasse zu implementieren hat. Neben der Zusammenführung der anderen Spielbasiselemente – die durch die Methoden `getSpielerA`, `getSpielerB`, `getRichter` und `steuereAblauf` repräsentiert wird – sollte das Spiel auch die weiteren Schnittstellen zur grafischen Oberfläche bereitstellen, die neben dem `SpielbrettView` noch erforderlich ist.
Um von dem Benutzer etwas einlesen zu können, definieren wir ein Interface `InputDialog`:

```
package framework;

public interface InputDialog {
    public String getEingabe(String aufforderung);
}
```

Um den Benutzer eine Information zu geben, definieren wir ein Interface `MessageDialog`:

```
package framework;

public interface MessageDialog {
    public void gebeAus(String nachricht);
}
```

Über die Methoden `getInputDialog` und `getMessageDialog` können andere Basiselemente auf die jeweiligen Objekte zugreifen.
Damit ist das Interface `Spiel` vollständig:

```
package framework;

public interface Spiel {
    public Spieler getSpielerA();
    public Spieler getSpielerB();
    public Schiedsrichter getRichter();
    public InputDialog getInputDialog();
    public MessageDialog getMessageDialog();
    public void steuereAblauf();
}
```

4.3 Ein Prototyp

Bevor wir mit dem auf der Klassenhierarchien für unser Framework beginnen, ist es ratsam, zuerst einen schlanken Prototyp zu entwicklen. Das bedeutet, wir versuchen für ein einfaches Spiel möglichest minimale Implementierungen für die definierten Interfaces zu finden. Einerseits kann dadurch getestet werden, ob die Klassen überhaupt so miteinander zusammenspielen können, wie wir uns das in der Entwurfsphase gedacht haben und andererseits können Ideen gesammelt werden für die spätere Entwicklung des Frameworks.

Einschub zur Namensgebung: Wir haben die Interfaces der Basiselemente nach den Objekten der realen Welt benannt. Damit haben wir aber auch das Problem, dass diese Namen nicht mehr für die Klassen zur Verfügung stehen, die sie implementieren. Dies ist kein Problem, wenn die Implementierung in eine bestimmte Richtung spezialisiert ist, wie z.B.

eine `SchachFigur` denkbar wäre, die `Figur` implementiert, sowie ein `SchachTurm` die von `SchachFigur erbt`. Wie nennen wir aber eine Klasse, die das Interface `Figur` minimal implementiert, ohne eine derartige Spezialisierung vorzunehmen?

In der Welt der Software-Entwickler haben sich für dieses Problem folgende Lösungen entwickelt, die zwar keine offiziellen Namenskonventionen darstellen, aber weit verbreitet sind[24]:

- Die einfachste Form ist das Anhängen von „Impl“ für Implementierung an den bestimmten Begriff. Im Falle des Interface `Figur` also `FigurImpl`. Damit sagen wir auch nichts weiter aus, als das die Klasse *eine* mögliche Implementierung des Interface ist.
- Für den Fall, dass die Klasse abstrakt ist, stehen zwei Möglichkeiten zur Verfügung: Wir könnten die betreffende Klasse `AbstrakteFigur` nennen oder `BasisFigur`. Diese Namensgebungen geben in gewisser Weise das Versprechen, dass sie gut geeignet sind, als Oberklassen konkreter Figurenklassen eingesetzt zu werden.
- Nehmen wir an, eine Klasse ist nicht abstrakt und implementiert das Interface so, dass anzunehmen ist, dass diese von allen Klienten verwendet werden kann, die keine Spezialisierungen benötigen, dann können wir sie `DefaultFigur` nennen.

Ich werde mich an diese Namensgebungen halten und in diesem Kapitel bei der Prototyp-Beschreibung die Klassen mit „Impl“ benennen, wenn noch nicht klar ist, welchen Zweck sie später im Framework erfüllen können.

In den folgenden Abschnitten werden Klassen vorgestellt, die unsere Interfaces so implementieren, dass das Spiel TicTacToe mit dem Hamstersimulator und zwei menschlichen Spielern spielbar ist.

4.3.1 Die Klasse SpielerImpl

Bei diesem Spiel muss ein Spieler seine Figuren nicht notwendigerweise kennen, das Instanzattribut `figuren` wird also nicht benötigt. deswegen wird es auf `null` gesetzt und nicht weiter verwendet. Ansonsten ist an dieser Klasse die Methode `getZug` interessant: Vom `Spiel` holt sie den `InputDialog` und vom `Schiedsrichter` den `Interpreter`, benutzt sie und gibt einen `Spielzug` zurück.
Ansonsten werden die `equals`- und `toString`-Methoden von `Object` überschrieben. Dies sollte eigentlich immer geschehen, wird aber aus Platzgründen in diesem Framework nur vorgenommen, wenn die Methoden auch benötigt werden.

```
package prototyp;

import framework.Spieler;
import framework.Figur;
import framework.Spiel;
import framework.InputDialog;
import framework.Interpreter;
import framework.Spielzug;
```

[24] Ein prominentes Beispiel ist die Java-Klassenbibliothek

```
public class SpielerImpl implements Spieler {

    private boolean istSpielerA;
    private Spiel spiel;
    private Figur[] figuren;

    public SpielerImpl(boolean istSpielerA) {
        this.istSpielerA = istSpielerA;
        this.figuren = null;
    }

    public boolean istSpielerA() {
        return istSpielerA;
    }

    public Spiel getSpiel() {
        return this.spiel;
    }

    public void setSpiel(Spiel spiel) {
        this.spiel = spiel;
    }

    public Figur[] getFiguren() {
        return this.figuren;
    }

    public void setFiguren(Figur[] figuren) {
        this.figuren = figuren;
    }

    public Spielzug getZug() {
        InputDialog dialog = this.spiel.getInputDialog();
        Interpreter interpreter = this.spiel.getRichter().getInterpreter();
        String eingabe = dialog.getEingabe(toString() +
                                        ", geben Sie bitte Ihren Zug ein.");
        return interpreter.getZug(eingabe);
    }

    public String toString() {
        return (istSpielerA) ? "Spieler A" : "Spieler B";
    }

    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
```

```
        if (!(obj instanceof Spieler)) {
            return false;
        }
        Spieler andererSpieler = (Spieler) obj;
        if (this.istSpielerA == andererSpieler.istSpielerA()) {
            return true;
        }
        return false;
    }
}
```

4.3.2 Die Klasse Sender

Wir haben schon im Entwurf betrachtet, dass der `SpielbrettView` dem `Empfaenger` im Beobachtermuster entspricht. Der zugehörige Sender ist das `Spielbrett`, dass von der Klasse `Sender` erbt. Diese Klasse können wir allgemein wiederverwenden, deswegen speichern wir sie nicht im `prototyp`-, sondern im `framework`-Paket.

```
package framework;

import java.util.ArrayList;
import java.util.List;

abstract public class Sender {

    List<SpielbrettView> empfaenger;

    public Sender() {
        this.empfaenger = new ArrayList<SpielbrettView>();
    }

    public void meldeAn(SpielbrettView view) {
        this.empfaenger.add (view);
    }

    protected void benachrichtige() {
        for (SpielbrettView view : empfaenger) {
            if (view != null) {
                view.aktualisiere((Spielbrett) this);
            }
        }
    }
}
```

4.3.3 Die Klasse SpielbrettImpl

Wie im vorgehenden Abschnitt schon erwähnt, ist das Spielbrett der Sender, also erbt die Klasse `SpielbrettImpl` von der Klasse `Sender`. Diese Klasse bietet mehr Funktionalität, als es das Interface `Spielbrett` vorsieht: Es geht davon aus, dass seine Spielfelder Koordinaten

haben und kann mittels der Methode `getFeld` über die Parametereingabe von Koordinaten das betreffende Spielfeld liefern. Dies ist notwendig für die Implementierung der Spielregeln im `TicTacToeSchiedsrichter` (siehe Abschnitt 4.3.7).

```
package prototyp;

import framework.Spielbrett;
import framework.Sender;
import framework.Spielfeld;
import framework.Schiedsrichter;
import framework.Figur;

public class SpielbrettImpl extends Sender implements Spielbrett {

    private Schiedsrichter richter;
    private Spielfeld[] felder;

    public SpielbrettImpl(Schiedsrichter richter) {
        this.richter = richter;
        this.felder = new Spielfeld[9];
        int zaehler = 0;
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                Spielfeld feld = new SpielfeldImpl(this, i, j);
                this.felder[zaehler] = feld;
                zaehler++;
            }
        }
    }

    public void setRichter(Schiedsrichter richter) {
        this.richter = richter;
    }

    public Schiedsrichter getRichter() {
        return this.richter;
    }

    public void setFelder(Spielfeld[] felder) {
        this.felder = felder;
    }

    public Spielfeld[] getFelder() {
        return this.felder;
    }

    public Spielfeld getFeld(int reihe, int spalte) {
        for (Spielfeld feld : this.felder) {
```

```
            SpielfeldImpl feldImpl = (SpielfeldImpl) feld;
            if (feldImpl.getReihe() == reihe
                    && feldImpl.getSpalte() == spalte) {
                return feldImpl;
            }
        }
        return null;
    }

    public void weiseZu(Spielfeld feld, Figur figur) {
        if (gehoertDazu(feld)) {
            feld.setFigur(figur);
            if (figur != null) {
                figur.setFeld(feld);
            }
        }
        benachrichtige();
    }

    public void wandleUm(Figur alteFigur, Figur neueFigur) {
        if (alteFigur != null && neueFigur != null) {
            weiseZu(alteFigur.getFeld(), neueFigur);
            alteFigur.setFeld(null);
        }
        benachrichtige();
    }

    public void initialisiere() { }

    public boolean gehoertDazu(Spielfeld pruefFeld) {
        if (pruefFeld == null) {
            return false;
        }
        for (Spielfeld feld : this.felder) {
            if (pruefFeld.equals(feld)) {
                return true;
            }
        }
        return false;
    }
}
```

4.3.4 Die Klasse SpielfeldImpl

Wie im vorhergehenden Abschnitt schon erwähnt, haben die Spielfelder Koordinaten. Diese werden im Konstruktor gesetzt und können danach sinnvollerweise von außen nicht mehr geändert werden. Ansonsten besteht diese Klasse nur aus `set`- und `get`-Methoden.

```
package prototyp;
```

```
import framework.Spielfeld;
import framework.Spielbrett;
import framework.Figur;

public class SpielfeldImpl implements Spielfeld{

    private Figur figur;
    private Spielbrett brett;
    private int reihe;
    private int spalte;

    public SpielfeldImpl (Spielbrett brett, int reihe, int spalte) {
        this.brett = brett;
        this.figur = null;
        this.reihe = reihe;
        this.spalte = spalte;
    }

    public void setFigur(Figur figur) {
        this.figur = figur;
    }

    public Figur getFigur() {
        return this.figur;
    }

    public Spielbrett getBrett() {
        return this.brett;
    }

    public int getReihe() {
        return this.reihe;
    }

    public int getSpalte() {
        return this.spalte;
    }
}
```

4.3.5 Die Klasse FigurImpl

TicTacToe gehört zu den Spielen, bei denen eine Figur inaktiv wird, sobald sie einmal gesetzt wird. Aus diesem Grunde reicht auch die minimale Implementierung des `Figur`-Interfaces für dieses Spiel aus:

```
package prototyp;

import framework.Figur;
```

```
import framework.Spieler;
import framework.Spielfeld;

public class FigurImpl implements Figur {

    private Spieler spieler;
    private Spielfeld feld;

    public FigurImpl(Spieler spieler, Spielfeld feld) {
        this.spieler = spieler;
        this.feld = feld;
    }

    public Spieler getSpieler() {
        return this.spieler;
    }

    public void setSpieler(Spieler spieler) {
        this.spieler = spieler;
    }

    public void setFeld(Spielfeld feld) {
        this.feld = feld;
    }

    public Spielfeld getFeld() {
        return this.feld;
    }
}
```

4.3.6 Die Klasse SpielzugImpl

Ein Spielzug besteht bei TicTacToe nur aus der Angabe eines einzelnen Feldes. Dementsprechend wird auch die Klasse `SpielzugImpl` implementiert:

```
package prototyp;

import framework.Spielzug;
import framework.Spielfeld;

public class SpielzugImpl implements Spielzug {

    private Spielfeld feld;

    public SpielzugImpl(Spielfeld feld) {
        this.feld = feld;
    }

    public Spielfeld getStart() {
```

```
        return this.feld;
    }

    public Spielfeld getZiel() {
        return this.feld;
    }

    public Spielfeld[] getFelder() {
        Spielfeld[] felder = { feld };
        return felder;
    }

    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (!(obj instanceof Spielzug)) {
            return false;
        }
        Spielzug andererZug = (Spielzug) obj;
        if (andererZug.getZiel().equals(this.feld)) {
            return true;
        }
        return false;
    }
}
```

4.3.7 Die Klasse TicTacToeSchiedsrichter

Die Klasse `TicTacToeSchiedsrichter` erhält deswegen diesen Namen, weil sie die erste ist, die wirklich spielabhängig ist. Es folgen einige Erläuterungen zu ihrer Implementierung:

- Die Methode `ermittleSpieler` ist so implementiert, dass Spieler A anfängt und dann immer abwechselnd gezogen wird.
- Ein Zug wird ausgeführt, in dem auf das im Zug angegebene Feld eine Figur des Spielers gestellt wird, der gerade an der Reihe ist (siehe `fuehreZugAus`).
- Jedes freie Feld auf dem Spielfeld stellt eine Zugmöglichkeit dar. Alle Zugmöglichkeiten der aktuellen Spielsituation liefert `getAlleZuege` zurück.
- Die Methode `zugKorrekt` überprüft, ob der angegebene Zug in den Zügen, die getAlleZuege ermittelt, enthalten ist.
- Die Hilfsmethoden `reiheVorhanden`, `spalteVorhanden` und `diagonaleVorhanden` melden an `spielBeendet`, ob ein Spieler es geschafft hat, drei Figuren in eine Reihe, Spalte oder Diagonale zu bringen.

```
package prototyp;
```

```
import java.util.List;
import java.util.ArrayList;
import framework.Schiedsrichter;
import framework.Spiel;
import framework.Spielbrett;
import framework.Interpreter;
import framework.Spieler;
import framework.Spielfeld;
import framework.Spielzug;
import framework.MessageDialog;
import framework.Figur;

public class TicTacToeSchiedsrichter implements Schiedsrichter {

    private Spiel spiel;
    private Spielbrett brett;
    private Spieler sieger;
    private Interpreter interpreter;

    public TicTacToeSchiedsrichter() {
        this.spiel = null;
        this.brett = new SpielbrettImpl(this);
        this.sieger = null;
        this.interpreter = new TicTacToeInterpreter(this);
    }

    public Schiedsrichter kopiere() {
        Schiedsrichter kopie = new TicTacToeSchiedsrichter();
        kopie.setSpiel() = this.spiel;
        kopie.setBrett() = this.brett;
    }

    public void setBrett(Spielbrett brett) {
        this.brett = brett;
    }

    public Spielbrett getBrett() {
        return this.brett;
    }

    public void setSpiel(Spiel spiel) {
        this.spiel = spiel;
    }

    public Spiel getSpiel() {
        return this.spiel;
    }
```

```
public Interpreter getInterpreter() {
    return this.interpreter;
}

public Spieler ermittleSpieler(Spieler vorherigerSpieler) {
    Spieler spielerA = this.spiel.getSpielerA();
    Spieler spielerB = this.spiel.getSpielerB();
    if (vorherigerSpieler == null
        || vorherigerSpieler.equals(spielerB)) {
        return spielerA;
    } else {
        return spielerB;
    }
}

public void fuehreZugAus(Spielzug zug, Spieler spieler) {
    Spielfeld feld = zug.getZiel();
    this.brett.weiseZu(feld, new FigurImpl(spieler, feld));
}

public boolean zugKorrekt(Spielzug pruefZug, Spieler spieler) {
    if (pruefZug == null) {
        this.sieger = (spieler.istSpielerA())
                       ? this.spiel.getSpielerB()
                       : this.spiel.getSpielerA();
        return false;
    }
    for (Spielzug zug : getAlleZuege(spieler)) {
        if (pruefZug.equals(zug)) {
            return true;
        }
    }
    this.sieger = (spieler.istSpielerA())
                   ? this.spiel.getSpielerB()
                   : this.spiel.getSpielerA();
    return false;
}

public Spielzug[] getAlleZuege(Spieler spieler) {
    List<Spielzug> zuege = new ArrayList<Spielzug>();
    for (Spielfeld feld : this.brett.getFelder()) {
        if (feld.getFigur() == null) {
            zuege.add (new SpielzugImpl(feld));
        }
    }
    return zuege.toArray(new Spielzug[zuege.size()]);
}
```

```
public boolean spielBeendet() {
    this.sieger = diagonaleVorhanden();
    if (this.sieger != null) {
        return true;
    }
    this.sieger = reiheVorhanden();
    if (this.sieger != null) {
        return true;
    }
    this.sieger = spalteVorhanden();
    if (this.sieger != null) {
        return true;
    }
    for (Spielfeld feld : this.brett.getFelder()) {
       if (feld.getFigur() == null) {
         return false;
       }
    }
    return true;
}

private Spieler reiheVorhanden() {
    Figur aktuelleFigur;
    SpielbrettImpl brettImpl = (SpielbrettImpl) this.brett;
    for (int i = 0; i < 3; i++) {
        int zaehler = 0;
        for (int j = 0; j < 3; j++) {
            aktuelleFigur = brettImpl.getFeld(i, j).getFigur();
            if (aktuelleFigur != null) {
                if(aktuelleFigur.getSpieler().istSpielerA()) {
                    zaehler++;
                } else {
                    zaehler--;
                }
            }
        }
        if (zaehler == 3) {
            return this.spiel.getSpielerA();
        } else if (zaehler == -3) {
            return this.spiel.getSpielerB();
        }
    }
    return null;
}

private Spieler spalteVorhanden() {
    Figur aktuelleFigur;
    SpielbrettImpl brettImpl = (SpielbrettImpl) this.brett;
```

```
    for (int i = 0; i < 3; i++) {
        int zaehler = 0;
        for (int j = 0; j < 3; j++) {
            aktuelleFigur = brettImpl.getFeld(j, i).getFigur();
            if (aktuelleFigur != null) {
                if (aktuelleFigur.getSpieler().istSpielerA()) {
                    zaehler++;
                } else {
                    zaehler--;
                }
            }
        }
        if (zaehler == 3) {
            return this.spiel.getSpielerA();
        } else if (zaehler == -3) {
            return this.spiel.getSpielerB();
        }
    }
    return null;
}

private Spieler diagonaleVorhanden() {
    int zaehler1 = 0;
    int zaehler2 = 0;
    Figur aktuelleFigur;
    SpielbrettImpl brettImpl = (SpielbrettImpl) this.brett;
    for (int i = 0; i < 3; i++) {
        aktuelleFigur = brettImpl.getFeld(i, i).getFigur();
        if (aktuelleFigur != null) {
            if (aktuelleFigur.getSpieler().istSpielerA()) {
                zaehler1++;
            } else {
                zaehler1--;
            }
        }
        aktuelleFigur = brettImpl.getFeld(i, 2 - i).getFigur();
        if (aktuelleFigur != null) {
            if (aktuelleFigur.getSpieler().istSpielerA()) {
                zaehler2++;
            } else {
                zaehler2--;
            }
        }
    }
    if (zaehler1 == 3 || zaehler2 == 3) {
        return this.spiel.getSpielerA();
    } else if (zaehler1 == -3 || zaehler2 == -3) {
        return this.spiel.getSpielerB();
```

```
        } else { //keine Diagonale vorhanden
            return null;
        }
    }

    public void verkuendeSieger() {
        MessageDialog dialog = this.spiel.getMessageDialog();
        if (sieger != null) {
            dialog.gebeAus(sieger.toString() + " ist Sieger!");
        } else {
            dialog.gebeAus("Das Spiel geht unentschieden zu Ende!");
        }
    }
}
```

4.3.8 Die Klasse TicTacToeInterpreter

Die Methode `getZug` der Klasse `TicTacToeInterpreter` geht davon aus, dass das Spielbrett von TicTacToe so nummeriert ist, wie der Nummernblock auf der Tastatur:

```
package prototyp;

import framework.Interpreter;
import framework.Schiedsrichter;
import framework.Spielzug;

public class TicTacToeInterpreter implements Interpreter {

    private Schiedsrichter richter;

    public TicTacToeInterpreter(Schiedsrichter richter) {
        this.richter = richter;
    }

    public Spielzug getZug(String str) {
        char c = str.charAt(0);
        SpielbrettImpl brettImpl =
                (SpielbrettImpl) this.richter.getBrett();
        switch (c) {
        case '1':
            return new SpielzugImpl(brettImpl.getFeld (2, 0));
        case '2':
            return new SpielzugImpl(brettImpl.getFeld (2, 1));
        case '3':
            return new SpielzugImpl(brettImpl.getFeld (2, 2));
        case '4':
            return new SpielzugImpl(brettImpl.getFeld (1, 0));
        case '5':
            return new SpielzugImpl(brettImpl.getFeld (1, 1));
```

```
        case '6':
            return new SpielzugImpl(brettImpl.getFeld (1, 2));
        case '7':
            return new SpielzugImpl(brettImpl.getFeld (0, 0));
        case '8':
            return new SpielzugImpl(brettImpl.getFeld (0, 1));
        case '9':
            return new SpielzugImpl(brettImpl.getFeld (0, 2));
        }
        return null;
    }
}
```

4.3.9 Die Klasse SpielbrettViewImpl

Die Klasse `SpielbrettViewImpl` bildet das Spielbrett im Hamstersimulator auf den ersten neun Kacheln links oben im Territorium ab. In einem Array vom Typ `Hamster` speichert sie für alle neun Kacheln, ob dort schon ein Hamster steht oder nicht.
In der Methode `aktualisiere` wird dann geprüft, ob ein auf dem Spielbrett vorhandener Hamster auch schon im Territorium steht. Wenn dies nicht der Fall ist, wird auf der entsprechenden Kachel ein neuer Hamster erzeugt. Die Hamster mit Blickrichtung `NORD` stellen die Figuren von Spieler A dar und die mit Blickrichtung `SUED` stellen die Figuren von Spieler B dar.

```
package prototyp;

import framework.SpielbrettView;
import framework.Spielbrett;
import framework.Spielfeld;

public class SpielbrettViewImpl implements SpielbrettView {

    private Hamster[] belegung;

    public SpielbrettViewImpl() {
        this.belegung = new Hamster[9];
    }

    public void aktualisiere(Spielbrett brett) {
        Spielfeld[] felder = brett.getFelder();
        for (int i = 0; i < this.belegung.length; i++) {
            SpielfeldImpl feldImpl = (SpielfeldImpl) felder[i];
            if (feldImpl.getFigur() != null && belegung[i] == null) {
                int reihe = feldImpl.getReihe();
                int spalte = feldImpl.getSpalte();
                int blick = (feldImpl.getFigur().getSpieler().istSpielerA())
                                ? Hamster.NORD
                                : Hamster.SUED;
                Hamster hamster = new Hamster(reihe, spalte, blick, 0);
```

```
                belegung[i] = hamster;
            }
        }
    }
}
```

4.3.10 Die Klasse HamsterInputDialog

Als `InputDialog` wird die `liesZeichenkette`-Methode des Standardhamsters benutzt. Diese Klasse können wir immer verwenden, wenn wir den Hamstersimulator als View benutzen, deswegen kommt sie in das Paket `hamsterView`.

```
package hamsterView;

import framework.InputDialog;

public class HamsterInputDialog implements InputDialog {

    public String getEingabe(String aufforderung) {
        return Hamster.getStandardHamster().liesZeichenkette(aufforderung);
    }
}
```

4.3.11 Die Klasse HamsterMessageDialog

Als `MessageDialog` wird die `schreib`-Methode des Standardhamsters benutzt. Diese Klasse können wir immer verwenden, wenn wir den Hamstersimulator als View benutzen, deswegen kommt sie in das Paket `hamsterView`.

```
package hamsterView;

import framework.MessageDialog;

public class HamsterMessageDialog implements MessageDialog {

    public void gebeAus(String nachricht) {
        Hamster.getStandardHamster().schreib(nachricht);
    }
}
```

4.3.12 Die Klasse SpielImpl

Die Klasse `SpielImpl` stellt sozusagen das Wurzelelement unser Basisklassen dar. Hier ist der Ablauf des Spiels mit Hilfe des Entwurfsmusters Schablonenmethode (siehe Kapitel 3.10) in der Methode `steuereAblauf` implementiert. Wie diese aufgebaut ist, wurde schon in Abschnitt 4.2.8 diskutiert.

```
package prototyp;

import java.util.List;
```

```
import java.util.ArrayList;
import framework.Spiel;
import framework.Spieler;
import framework.Schiedsrichter;
import framework.SpielbrettView;
import framework.InputDialog;
import framework.MessageDialog;
import framework.Figur;
import framework.Spielzug;

public class SpielImpl implements Spiel {

    private Spieler spielerA;
    private Spieler spielerB;
    private Schiedsrichter richter;
    private SpielbrettView view;
    private InputDialog inputDialog;
    private MessageDialog messageDialog;

    public SpielImpl(Spieler spielerA, Spieler spielerB,
                     Schiedsrichter richter, SpielbrettView view,
                     InputDialog inputDialog, MessageDialog messageDialog) {

        this.spielerA = spielerA;
        this.spielerA.setSpiel(this);
        this.spielerB = spielerB;
        this.spielerB.setSpiel(this);
        this.richter = richter;
        this.richter.setSpiel(this);
        this.inputDialog = inputDialog;
        this.messageDialog = messageDialog;

        this.view = view;
        richter.getBrett().meldeAn(view);
    }

    public Spieler getSpielerA() {
        return this.spielerA;
    }

    public Spieler getSpielerB() {
        return this.spielerB;
    }

    public Schiedsrichter getRichter() {
        return this.richter;
    }
```

```
    public InputDialog getInputDialog() {
        return this.inputDialog;
    }

    public MessageDialog getMessageDialog() {
        return this.messageDialog;
    }

    public void steuereAblauf() {
        Spieler aktuellerSpieler = null;
        Figur figur;
        Spielzug zug;
        boolean zugKorrekt;
        this.messageDialog.gebeAus("Willkommen beim TicTacToe-Spiel!");
        do {
            aktuellerSpieler =
                    this.richter.ermittleSpieler(aktuellerSpieler);
            zug = aktuellerSpieler.getZug();
            if (!richter.zugKorrekt(zug, aktuellerSpieler)) {
                break;
            }
            richter.fuehreZugAus(zug, aktuellerSpieler);
        } while (!(richter.spielBeendet()));
        richter.verkuendeSieger();
    }
}
```

4.3.13 Ein TicTacToe-Hamsterprogramm

Im Hauptprogramm müssen wir alle Objekte, die die Klasse `SpielImpl` als Parameter benötigt erzeugen, dann ein Objekt dieser Klasse selbst erzeugen und schließlich muss nur noch die Methode `steuereAblauf` aufgerufen werden.
Wenn der CLASSPATH im Hamstersimulator auf das Verzeichnis „spieleFramework“ gesetzt ist und ein Territorium von mindestens drei mal drei Kacheln zur Verfügung steht, ist das Spiel TicTacToe damit spielbar.

```
import framework.Spieler;
import framework.Schiedsrichter;
import framework.SpielbrettView;
import framework.InputDialog;
import framework.MessageDialog;
import framework.Spiel;
import prototyp.SpielerImpl;
import prototyp.TicTacToeSchiedsrichter;
import prototyp.SpielbrettViewImpl;
import prototyp.SpielImpl;
import prototyp.HamsterInputDialog;
import prototyp.HamsterMessageDialog;
```

```
void main() {
    Spieler spielerA = new SpielerImpl(true);
    Spieler spielerB = new SpielerImpl(false);
    Schiedsrichter richter = new TicTacToeSchiedsrichter();
    SpielbrettView view = new SpielbrettViewImpl();
    InputDialog input = new HamsterInputDialog();
    MessageDialog message = new HamsterMessageDialog();

    Spiel spiel = new SpielImpl(spielerA, spielerB, richter, view,
                                input, message);

    spiel.steuereAblauf();
}
```

4.4 Implementierung

4.4.1 Das Spielfeld

Als konkrete Spielfeldklasse stellt das Framework zunächst als minimale Implementierung von `Spielfeld` das `DefaultSpielfeld` bereit.

```
package framework;

import java.util.List;
import java.util.ArrayList;

public class DefaultSpielfeld implements Spielfeld {

    private Spielbrett brett;
    private Figur figur;

    public DefaultSpielfeld(Spielbrett brett) {
        this.brett = brett;
        this.figuren = new ArrayList<Figur>();
    }

    public Spielbrett getBrett() {
        return this.brett;
    }

    public void setBrett(Spielbrett brett) {
        this.brett = brett;
    }

    public Figur getFigur() {
        return this.figur;
    }

    public void setFigur(Figur figur) {
```

```
        this.figur = figur;
    }
}
```

Die Klasse `KoordinatenSpielfeld` erbt nun von `DefaultSpielfeld` und wird um Koordinaten erweitert, da wir bei der Prototypenentwicklung gesehen haben, dass schon das relativ simple Spiele wie TicTacToe diese für die Spielregelimplementierung benötigt.
Dies wird erreicht durch die Instanzattribute `reihe` und `spalte`. Der Konstruktor wird entsprechend angepasst, zudem werden `get`-Methoden bereitgestellt.
Die `toString`-Methode macht nun Folgendes: Sie „übersetzt“ die Koordinaten auf die Schreibweise des Schachbretts. Sie geht davon aus, dass das Feld (0,0) dem Feld (A,1) entspricht.
Da ein `KoordinatenSpielfeld` nur zu einem `KoordinatenSpielbrett` passt, wird durch das Werfen einer `IllegalArgumentException` im Konstruktor verhindert, dass solch ein Spielfeld mit einem Spielbrett kombiniert wird, dass keine Koordinaten kennt.

```
package framework;

public class KoordinatenSpielfeld extends DefaultSpielfeld {

    private int reihe;
    private int spalte;

    public KoordinatenSpielfeld(Spielbrett brett, int reihe, int spalte) {
        super(brett);
        if (!(brett instanceof KoordinatenSpielbrett)) {
            throw new IllegalArgumentException();
        }
        this.reihe = reihe;
        this.spalte = spalte;
    }

    public int getReihe() {
        return this.reihe;
    }

    public int getSpalte() {
        return this.spalte;
    }

    public String toString() {
        String x = new Integer(this.reihe + 1).toString();
        char y;
        if (spalte < 26) {
            y = (char) (this.spalte + 97);
        } else {
            y = (char) (this.spalte + 39);
        }
        return y + x;
    }
```

```
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (!(obj instanceof KoordinatenSpielfeld)) {
            return false;
        }
        KoordinatenSpielfeld anderesFeld = (KoordinatenSpielfeld) obj;
        if (this.reihe == anderesFeld.getReihe()
            && this.spalte == anderesFeld.getSpalte()) {
            return true;
        }
        return false;
    }
}
```

4.4.2 Der Spielzug

Aus den Überlegungen zum Interface `Spielzug` (Kapitel 4.2.5, Seite 128) geht hervor, dass wir drei konkrete Spielzugklassen benötigen. Hinzu kommt, dass für die Implementierung des `MinimaxSpieler` (siehe Abschnitt 4.4.5) jeder Spielzug einen Wert haben muss.
Es gibt mehrere Möglichkeiten, diese Anforderung umzusetzen: Eine wäre, das `Spielzug`-Interface um die Methoden `getWert` und `setWert` zu erweitern und sie entsprechend in allen Spielzugklassen zu implementieren. Dies hätte jedoch zwei Nachteile:

- Es wären keine Spielzüge ohne Wert möglich. Das ist deswegen ein Nachteil, weil es das Framework unflexibler machen würde.
- Die set- und get-Methoden für den Wert und das zugehörige Instanzattribut werden nicht wiederverwendet, sondern in allen drei Klassen fest codiert.

Wir wollen deswegen eine flexiblere Lösung wählen: Eine abstrakte Klasse `WertSpielzug` implementiert `Spielzug`. Von ihr werden die drei konkreten Spielzugklassen abgeleitet. Damit bleibt einerseits die oberste Schnittstelle minimal und andererseits wird der Quellcode für den Wert wiederverwendet.
Der Nachteil dieser Lösung besteht darin, dass zwar unbewertete Spielzüge generell möglich sind, diese müssten aber dann neu implementiert werden und der Code der bereits existierenden Klassen könnte dafür *nicht* wiederverwendet werden.

```
package framework;

abstract public class WertSpielzug implements Spielzug {

    private int wert = 0;

    public int getWert() {
        return this.wert;
    }
```

```
    public void setWert(int wert) {
        this.wert = wert;
    }
}
```

Wenden wir uns nun den drei konkreten Spielzugklassen zu. Eine Klasse `EinerSpielzug` wird für Spiele wie Reversi, TictacToe oder Vier Gewinnt zur Verfügung gestellt. Diese Art von Spielen hat gemeinsam, dass jeder Spieler nur eine Sorte Figuren hat und ein Zug darin besteht, eine neue Figur auf ein Feld des Spielbretts zu stellen.
In Abschnitt 4.2.5 wurde bereits erläutert, warum diese Klasse auch die Methoden `getStart` und `getFelder` implementieren muss, obwohl `getZiel` für sie ausreicht. Es soll nun geklärt werden, wie wir sie implementieren können:

- Für `getStart` gibt es zwei Möglichkeiten: Sie könnte `null` zurückgeben. Dies wäre zwar auf den ersten Blick intuitiv, weil es bei so einem Spielzug kein Startfeld gibt, denn die Figur kommt von außerhalb des Brettes dazu. Andererseits könnte das zu unnötigen `NullPointerException`s führen, wenn die Methode an Stelle der `getZiel` Methode verwendet wird. Aus diesem Grunde ist es besser, wenn sie dasselbe Feld zurückgibt wie `getZiel`.
- Der einzig sinnvolle Rückgabewert für `getFelder` in dieser Klasse ist ein Array, dass genau ein Element enthält, und zwar wieder genau das Spielfeld um das es in diesem Zug geht.

Es folgt nun der Quellcode für diese Klasse:

```
package framework;

public class EinerSpielzug extends WertSpielzug {

    private Spielfeld feld;

    public EinerSpielzug(Spielfeld feld) {
        this.feld = feld;
    }

    public Spielfeld getStart() {
        return this.feld;
    }

    public Spielfeld getZiel() {
        return this.feld;
    }

    public Spielfeld[] getFelder() {
        Spielfeld[] felder = { this.feld };
        return felder;
    }
```

```
    public String toString() {
        return "EinerSpielzug: " + this.feld;
    }

    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (!(obj instanceof Spielzug)) {
            return false;
        }
        Spielzug andererZug = (Spielzug) obj;
        if (andererZug.getFelder().length != 1) {
            return false;
        }
        if (this.feld.equals(andererZug.getFelder()[0])) {
            return true;
        }
        return false;
    }
}
```

Die Klasse `ZweierSpielzug` deckt die Anforderungen von Spielen wie Schach, Dame und Alapo ab. Die Implementierung der Methoden `getStart` und `getZiel` ist für diese Klasse trivial und `getFelder` gibt parallel zur Implementierung im `EinerSpielzug` ein Array mit dem Start- und dem Zielfeld zurück.

```
package framework;

public class ZweierSpielzug extends WertSpielzug {

    private Spielfeld start;
    private Spielfeld ziel;

    public ZweierSpielzug(Spielfeld start, Spielfeld ziel) {
        this.start = start;
        this.ziel = ziel;
    }

    public Spielfeld getStart() {
        return this.start;
    }

    public Spielfeld getZiel() {
        return this.ziel;
    }

    public Spielfeld[] getFelder() {
        Spielfeld[] felder = { this.start, this.ziel };
```

```
        return felder;
    }

    public String toString() {
        return this.start.toString() + ", " + this.ziel.toString();
    }

    public boolean equals (Object obj) {
        if (obj == null) {
            return false;
        }
        if (!(obj instanceof Spielzug)) {
            return false;
        }
        Spielzug andererZug = (Spielzug) obj;
        if (andererZug.getFelder().length != 2) {
            return false;
        }
        Spielfeld andererStart = andererZug.getStart();
        Spielfeld anderesZiel = andererZug.getZiel();
        if (this.start == null && andererStart == null) {
            if (this.ziel == null && anderesZiel == null) {
                return true;
            }
            if (this.ziel != null && anderesZiel != null
                && this.ziel.equals (anderesZiel)) {
                return true;
            }
            return false;
        } else if (this.start != null && andererStart != null
                    && this.start.equals (andererStart)) {
            if (this.ziel == null && anderesZiel == null) {
                return true;
            }
            if (this.ziel != null && anderesZiel != null
                && this.ziel.equals (anderesZiel)) {
                return true;
            }
            return false;
        }
        return false;
    }
}
```

Die Klasse `PolySpielzug` ist für Sonderfälle einzusetzen, wie zum Beispiel bei Mehrfachsprüngen im Damespiel. Sie verwaltet alle ihr zugeordneten Felder in einer `LinkedList`. Diese hat gegenüber der in dieser Arbeit hauptsächlich eingesetzten `ArrayList`, dass sie die Methoden `getFirst` und `getLast` anbietet. Dies können wir hier für `getStart` und `getZiel`

ausnutzen.

```
package framework;

import java.util.LinkedList;

public class PolySpielzug extends WertSpielzug {

    private LinkedList<Spielfeld> list;

    public PolySpielzug() {
        this.list = new LinkedList<Spielfeld>();
    }

    public PolySpielzug(Spielfeld start) {
        this.list = new LinkedList<Spielfeld>();
        this.list.add (start);
    }

    public PolySpielzug(LinkedList<Spielfeld> list) {
        this.list = list;
    }

    public Spielfeld getStart() {
        return (this.list.size() != 0)
                 ? (Spielfeld) this.list.getFirst()
                 : null;
    }

    public Spielfeld getZiel() {
        return (this.list.size() != 0)
                  ? (Spielfeld) this.list.getLast()
                  : null;
    }

    public Spielfeld[] getFelder() {
        return this.list.toArray(new Spielfeld[this.list.size()]);
    }

    public void erweitere(Spielfeld spielfeld) {
        this.list.add(spielfeld);
    }

    public String toString() {
        String erg = "";
        for (Spielfeld feld : list) {
            erg += " " + feld.toString() + ", ";
        }
```

```
        erg = erg.substring(0, erg.length() - 2);
        return erg;
    }

    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (!(obj instanceof Spielzug)) {
            return false;
        }
        Spielzug zug = (Spielzug) obj;
        int length_a = zug.getFelder().length;
        int length_b = this.getFelder().length;
        if (length_a  != length_b) {
            return false;
        }
        for (int i = 0; i < zug.getFelder().length; i++) {
            if (zug.getFelder()[i] != null) {
                if (!(zug.getFelder()[i].equals(this.getFelder()[i]))) {
                    return false;
                }
            } else {
                if (this.getFelder()[i] != null) {
                    return false;
                }
            }
        }
        return true;
    }
}
```

4.4.3 Das Spielbrett

Das Framework stellt mit der abstrakten Klasse `DefaultSpielbrett` eine Basisimplementierung bereit, auf der konkrete Spielbrettklassen aufbauen können.

```
package framework;

abstract public class DefaultSpielbrett extends Sender
                                        implements Spielbrett {

    protected Spielfeld[] felder;
    protected Schiedsrichter richter;

    public DefaultSpielbrett(Schiedsrichter richter) {
        super();
        this.richter = richter;
    }
```

```
public Schiedsrichter getRichter() {
    return this.richter;
}

public void setRichter(Schiedsrichter richter) {
    this.richter = richter;
}

public Spielfeld[] getFelder() {
    return this.felder;
}

public void setFelder(Spielfeld[] felder) {
    this.felder = felder;
}

public void weiseZu(Spielfeld feld, Figur figur) {
    if (figur != null) {
        figur.getFeld().setFigur(null);
    }
    feld.setFigur(figur);
    if (figur != null) {
        figur.setFeld(feld);
    }
    benachrichtige();
}

public void wandleUm(Figur alteFigur, Figur neueFigur) {
    Spielfeld feld = alteFigur.getFeld();
    alteFigur.setFeld(null);
    feld.setFigur(neueFigur);
    neueFigur.setFeld(feld);
    benachrichtige();
}

public boolean gehoertDazu(Spielfeld pruefFeld) {
    if (pruefFeld == null) {
        return false;
    }
    for (Spielfeld feld : this.felder) {
        if (pruefFeld.equals(feld)) {
            return true;
        }
    }
    return false;
}
public void initialisiere() { }
```

```
}
```

Parallel zum `KoordinatenSpielfeld` wird das `KoordinatenSpielbrett` bereitgestellt, um dem Spielbrett über Reihen und Spalten Koordinaten geben zu können. Mit der Methode `sucheFeld` ist es möglich, durch Angabe von Koordinaten ein `KoordinatenSpielfeld`-Objekt zu erhalten.

```
package framework;

import java.util.ArrayList;
import java.util.List;

abstract public class KoordinatenSpielbrett extends DefaultSpielbrett {

    protected int reihen;
    protected int spalten;

    public KoordinatenSpielbrett(Schiedsrichter richter,
                                 int reihen, int spalten) {
        super(richter);
        List<Spielfeld> list = new ArrayList<Spielfeld>();
        for (int i = 0; i < reihen; i++) {
            for (int j = 0; j < spalten; j++) {
                list.add (new KoordinatenSpielfeld (this, i, j));
            }
        }
        this.felder = list.toArray(new KoordinatenSpielfeld[list.size()]);
        this.reihen = reihen;
        this.spalten = spalten;
    }

    public KoordinatenSpielfeld sucheFeld(int reihe, int spalte) {
        for (Spielfeld feld : this.felder) {
            KoordinatenSpielfeld xyfeld = (KoordinatenSpielfeld) feld;
            if (xyfeld.getReihe() == reihe
                    && xyfeld.getSpalte() == spalte) {
                return xyfeld;
            }
        }
        return null;
    }
}
```

4.4.4 Die Figur

Das Framework stellt mit der Klasse `BasisKoordinatenFigur` eine Hilfsklasse für Spiele zur Verfügung, die Spielbretter und -felder mit Koordinaten verwenden und komplexe Spielregeln haben.

In Spielen wie Dame oder Schach ist es sinnvoll, in den Figuren die Regeln für die Spielzüge, die sie machen dürfen, zu implementieren, um die `Schiedsrichter`-Klasse zu entlasten.
Diese Figurenklassen können nun von dieser hier vorgestellten Klasse abgeleitet werden. Sie stellt einige häufig nachgefragte Funktionalitäten bereit, bspw. ob auf einem bestimmten Feld eine gegnerische Figur steht usw.
Zu den Methoden `istA`, `getX`, `getY` ist zu bemerken, dass sich durch sie zwar Regeln schneller implementieren lassen, aber der Code dadurch auch kryptischer wird. Es soll an dieser Stelle zwar generell die Idee solcher Abkürzungen aufgezeigt, aber gleichzeitig vor ihnen gewarnt werden.

```
package framework;

import java.util.List;
import java.util.ArrayList;

public class BasisKoordinatenFigur implements Figur {

    protected Spieler spieler;
    protected KoordinatenSpielfeld feld;

    public BasisKoordinatenFigur (Spieler spieler,
                                  KoordinatenSpielfeld feld) {
        this.spieler = spieler;
        this.feld = feld;
        Figur[] figuren = this.spieler.getFiguren();
        Figur[] figurenNeu = new Figur[figuren.length + 1];
        for (int i = 0; i < figuren.length; i++) {
            figurenNeu[i] = figuren[i];
        }
        figurenNeu[figuren.length] = this;
        this.spieler.setFiguren(figurenNeu);
    }

    public void setFeld(Spielfeld feld) {
        this.feld = (KoordinatenSpielfeld) feld;
    }

    public KoordinatenSpielfeld getFeld() {
        return this.feld;
    }

    public void setSpieler(Spieler spieler) {
        this.spieler = spieler;
    }

    public Spieler getSpieler() {
        return spieler;
    }
```

```
public boolean istA() {
    return this.spieler.istSpielerA();
}

protected int getX() {
    return this.feld.getReihe();
}

protected int getY() {
    return this.feld.getSpalte();
}

protected Spielzug pruefeFeld(int reihe, int spalte) {
    KoordinatenSpielbrett brett =
            (KoordinatenSpielbrett) this.feld.getBrett();
    Spielfeld ziel = brett.sucheFeld(reihe, spalte);
    if (ziel == null) {
        return null;
    }
    BasisKoordinatenFigur figur =
            (BasisKoordinatenFigur) ziel.getFigur();
    if (figur == null || (figur != null
                          && figur.istA() != this.istA())) {
        return new ZweierSpielzug(this.feld, ziel);
    }
    return null;
}

protected boolean feldFeindFigur(int reihe, int spalte) {
    KoordinatenSpielbrett brett =
            (KoordinatenSpielbrett) this.feld.getBrett();
    Spielfeld ziel = brett.sucheFeld(reihe, spalte);
    if (ziel == null) {
        return false;
    }
    BasisKoordinatenFigur figur =
            (BasisKoordinatenFigur) ziel.getFigur();
    if (figur != null && figur.istA() != this.istA()) {
        return true;
    }
    return false;
}

protected boolean feldFrei(int reihe, int spalte) {
    KoordinatenSpielbrett brett =
            (KoordinatenSpielbrett) this.feld.getBrett();
    Spielfeld ziel = brett.sucheFeld(reihe, spalte);
```

```
        if (ziel == null) {
            return false;
        }
        if (ziel.getFigur() == null) {
            return true;
        }
        return false;
    }

    public List<Spielzug> eineRichtung(int x, int y) {
        List<Spielzug> list = new ArrayList<Spielzug>();
        while (feldFrei(getX() + x, getY() + y)) {
            Spielzug zug = pruefeFeld(getX() + x, getY() + y);
            if (zug != null) {
                list.add(zug);
            }
            if (x > 0) {
                x++;
            }
            if (x < 0) {
                x--;
            }
            if (y > 0) {
                y++;
            }
            if (y < 0) {
                y--;
            }
        }
        Spielzug zug = pruefeFeld(getX() + x, getY() + y);
        if (zug != null) list.add(zug);
        return list;
    }
}
```

Außerdem wird mit dem Interface `BasisKonstanten` eine Koordinatensammlung zur Verfügung gestellt, die die Regelimplementierung ebenfalls erleichtern soll.
Die beiden Konstanten, die bisher definiert sind, stellen nur einen Anfang dar – man kann mit ihnen mit Hilfe von Schleifendurchläufen die Felder ansteuern, die diagonal mit dem aktuellen benachbart sind. Wie dies genau funktioniert, werden wir im Kapitel über das DameSpiel sehen.

```
package framework;

public interface BasisKonstanten {

    public static final int[][] DIAGONAL_1 =
            {{1, 1}, {1, -1}, {-1, 1}, {-1, -1}};
    public static final int[][] DIAGONAL_2 =
```

```
            {{2, 2}, {2, -2}, {-2, 2}, {-2, -2}};
}
```

4.4.5 Der Spieler

Die Klassenhierarchie zum Interface `Spieler` ist wie folgt aufgebaut: Es gibt eine Basisimplementierung `AbstrakterSpieler`, die die Anforderungen realisiert, dass ein Spieler das Spiel kennen soll, an dem er teilnimmt, sowie ob er Spieler A oder Spieler B ist. Davon erben die Klassen `MenschSpieler`, `SimplerComputerSpieler` und `MinimaxSpieler`.

```
package framework;

abstract public class AbstrakterSpieler implements Spieler{

    protected boolean istSpielerA;
    protected Spiel spiel;
    protected Figur[] figuren;

    public AbstrakterSpieler(boolean istSpielerA) {
        this.istSpielerA = istSpielerA;
        this.figuren = null;
    }

    public Figur[] getFiguren() {
        return this.figuren;
    }

    public void setFiguren(Figur[] figuren) {
        this.figuren = figuren;
    }

    public Spiel getSpiel() {
        return this.spiel;
    }

    public void setSpiel(Spiel spiel) {
        this.spiel = spiel;
    }

    public boolean istSpielerA() {
        return this.istSpielerA;
    }

    public String toString() {
        return this.istSpielerA ? "Spieler A" : "Spieler B";
    }

    public boolean equals(Object obj) {
        if (obj == null) {
```

```
            return false;
        }
        if (!(obj instanceof Spieler)) {
            return false;
        }
        Spieler andererSpieler = (Spieler) obj;
        if (this.istSpielerA == andererSpieler.istSpielerA()) {
            return true;
        }
        return false;
    }
}
```

Bevor wir uns mit der Klasse `MenschSpieler` auseinandersetzen, müssen wir auf die Implementierung des `Interpreter`-Interface eingehen:
Die Standardimplementierung ist mit dem `SpielzugInterpreter` gegeben. Dieser erwartet einen String in der Form [Spielfeld-Stringrepräsentation Trennzeichen Spielfeld-Stringrepräsentation usw.] und gibt einen entsprechenden Spielzug zurück. Je nach Zahl der Spielfelder einen `EinerSpielzug`, `ZweierSpielzug` oder `PolySpielzug`.

```
package framework;

import java.util.ArrayList;
import java.util.List;
import java.util.LinkedList;
import java.util.StringTokenizer;

public class SpielzugInterpreter implements Interpreter {

    private Schiedsrichter richter;

    public SpielzugInterpreter(Schiedsrichter richter) {
        this.richter = richter;
    }

    public Spielzug getZug(String str) {
        if (str == null) {
            return null;
        }
        LinkedList<Spielfeld> list = new LinkedList<Spielfeld>();
        StringTokenizer tokenizer =
                new StringTokenizer(str, ",;.:-_\t\n\r\f+*~#'?!/§$%&=");
        Spielfeld[] felder = richter.getBrett().getFelder();

        while (tokenizer.hasMoreTokens()) {
            String token = tokenizer.nextToken();
            boolean korrekt = false;
            for (Spielfeld feld : felder) {
                if (token.equals(feld.toString())) {
```

```
                    list.add(feld);
                    korrekt = true;
                    break;
                }
            }
            if (!korrekt) {
                return null;
            }
        }
        if (list.size() == 1) {
            return new EinerSpielzug(list.get(0));
        } else if (list.size() > 2) {
            return new PolySpielzug(list);
        } else {
            return new ZweierSpielzug(list.get(0), list.get(1));
        }
    }
}
```

In Abschnitt 4.2.6 wurde diskutiert, dass von Spiel zu Spiel unterschiedlich „Spezialzüge“ wie zum Beispiel das Aufgeben möglich sein können. Um dies flexibel gestalten zu können setzen wir das Dekorierer-Muster ein.
Die abstrakte Dekoriererklasse `InterpreterDekorierer` ist deckungsgleich mit der entsprechenden Klasse aus dem Hamsterbeispiel (siehe Kapitel 3.11.5, Seite 116):

```
package framework;

abstract public class InterpreterDekorierer implements Interpreter {

    Interpreter interpreter;

    public InterpreterDekorierer(Interpreter interpreter) {
        this.interpreter = interpreter;
    }

    public Spielzug getZug(String str) {
        return this.interpreter.getZug(str);
    }
}
```

Vom Framework wird ein konkreter Dekorierer angeboten:
Der AufgabeDekorierer erweitert den Interpreter um die Möglichkeit, die Spielaufgabe eines Spielers anzunehmen. Wenn ein Spieler den Kleinbuchstaben „x“ eingibt, gibt er damit auf.

```
package framework;

public class AufgabeDekorierer extends InterpreterDekorierer {

    public final static Spielzug AUFGABE_SPIELZUG = new EinerSpielzug(null);
```

```
    public AufgabeDekorierer(Interpreter interpreter) {
        super(interpreter);
    }

    public Spielzug getZug(String str) {
        Spielzug zug = super.getZug(str);
        if (zug == null) {
            zug = getAufgabeZug(str);
        }
        return zug;
    }

    public Spielzug getAufgabeZug(String str) {
        if (str.equals("x")) {
            return AUFGABE_SPIELZUG;
        }
        return null;
    }
}
```

Die Klasse `MenschSpieler` stellt die Schnittstelle zum Benutzer da. Sie entspricht im Prinzip der Klasse `SpielerImpl` des Prototypen, nur dass hier einige Methoden weggelassen werden können, die schon in `AbstrakterSpieler` implementiert sind.

```
package framework;

public class MenschSpieler extends AbstrakterSpieler {

    public MenschSpieler(boolean istSpielerA) {
        super(istSpielerA);
    }

    public Spielzug getZug() {
        InputDialog dialog = this.spiel.getInputDialog();
        Interpreter interpreter = this.spiel.getRichter().getInterpreter();
        String eingabe = dialog.getEingabe(toString() +
                                        ", geben Sie bitte Ihren Zug ein.");
        return interpreter.getZug(eingabe);
    }
}
```

Die Klasse `SimplerComputerSpieler` stellt eine einfache Spielerimplementierung zur Verfügung, die in der Methode `getZug` aus allen gerade möglichen Spielzügen per Zufallsgenerator einen auswählt und diesen zurückgibt.

```
package framework;

public class SimplerComputerSpieler extends AbstrakterSpieler {
```

```
    public SimplerComputerSpieler(boolean istSpielerA) {
        super(istSpielerA);
    }

    public Spielzug getZug() {
        Spielzug[] zuege = this.spiel.getRichter.getAlleZuege(this);
        int zufall = (int) (Math.random() * zuege.length);
        return zuege[zufall];
    }
}
```

Bevor als nächstes die Implementierung des Minimax-Algorithmus vorgestellt werden kann, muss einer weitere Hilfsklasse vorgestellt werden: Eine `Bewertung` kann von Spielalgorithmen benutzt werden, um eine aktuelle Spielsituation zu bewerten.
Das Interface `Bewertung` stellt die oberste Schnittstelle für die Bewertungen von Spielsituationen dar. Es fordert die Implementierung einer Methode `analysiere`, die einen `int`-Wert zurückgibt. Ein positiver Wert steht für einen Vorteil für Spieler A, ein negativer Wert für einen Vorteil für den Spieler B – jeweils natürlich nur auf die aktuelle Spielsituation bezogen. Der Betrag des Wertes stellt die Größe des jeweiligen Vorteils dar.

```
package framework;

public interface Bewertung {
    public int analysiere(Spielbrett brett);
}
```

Die Klasse `DefaultBewertung` stellt die einfachst mögliche Implementierung von `Bewertung` bereit: Es werden einfach die Figuren der Spieler gezählt und das Verhältnis dieser Anzahlen zurückgegeben. Diese Klasse kann als Oberklasse für bessere Bewertungen dienen, die abhängig vom konkreten Spiel genauere Analysen vornehmen können.

```
package framework;

public class DefaultBewertung implements Bewertung {

    public int analysiere(Spielbrett brett) {
        int ergebnis = 0;
        //Anzahl der Figuren
        Spielfeld[] felder = brett.getFelder();
        for (Spielfeld feld : felder) {
            Figur figur = feld.getFigur();
            if (figur != null) {
                if (figur.getSpieler().istSpielerA()) {
                    ergebnis++;
                } else {
                    ergebnis--;
                }
            }
```

```
        }
        return ergebnis;
    }
}
```

Der `MinimaxSpieler` ist eine Spielerimplementierung, die den Minimax-Algorithmus verwendet. Dieser wird gut verständlich in [BB04] eingeführt. Die hier vorgestellte Klasse benutzt eine in einem `Bewertung`-Objekt gekapselte Bewertungsfunktion. Damit wird das Strategie-Muster verwendet, da diese flexibel ausgetauscht werden kann.

```
package framework;

import java.util.ArrayList;

public class MinimaxSpieler extends AbstrakterSpieler {

    public final static int MINIMUM = - 1000000;
    public final static int MAXIMUM = 1000000;

    private int spielStaerke;
    private Bewertung bewertung;

    public MinimaxSpieler(boolean istSpielerA, int spielStaerke,
                          Bewertung bewertung) {
        super(istSpielerA);
        this.spielStaerke = spielStaerke;
        this.bewertung = bewertung;
    }

    public Spielzug getZug() {
        Schiedsrichter richter = this.spiel.getRichter();
        Spielbrett brett = richter.getBrett();
        Spielzug naechsterZug = null;
        if (this.istSpielerA()) {
            naechsterZug = maxWert(this.spielStaerke, brett, richter);
        } else {
            naechsterZug = minWert(this.spielStaerke, brett, richter);
        }
        PolySpielzug poly = new PolySpielzug();
        for (Spielfeld feld : naechsterZug.getFelder()) {
            poly.erweitere(
                transformiere(feld, feld.getBrett(), richter.getBrett())
            );
        }
        return poly;
    }

    public WertSpielzug maxWert(int restTiefe, Spielbrett brett,
                            Schiedsrichter richter) {
```

```
    int ermittelt = MINIMUM;
    WertSpielzug besterZug = null;
    Spielzug[] moeglicheSpielzuege =
            richter.getAlleZuege(this.spiel.getSpielerA());
    for (Spielzug z : moeglicheSpielzuege) {
        Schiedsrichter kopie = richter.kopiere();
        PolySpielzug poly = new PolySpielzug();
        for (Spielfeld feld : z.getFelder()) {
            poly.erweitere(
                transformiere(feld, richter.getBrett(), kopie.getBrett())
            );
        }
        kopie.fuehreZugAus(poly, spiel.getSpielerA());
        int zugWert = 0;
        if (restTiefe <= 1
            || keinZugMoeglich(kopie, spiel.getSpielerA())) {
            zugWert = bewertung.analysiere(kopie.getBrett());
        } else {
            WertSpielzug zug =
                minWert(restTiefe - 1, kopie.getBrett(), kopie);
            zugWert = (zug != null) ? zug.getWert() : 0;
        }
        int zufall = (int) (Math.random() * 2);
        if (zugWert > ermittelt
            || (zugWert == ermittelt && zufall == 0)) {
            ermittelt = zugWert;
            besterZug = poly;
            besterZug.setWert(zugWert);
        }
    }
    return besterZug;
}

public WertSpielzug minWert(int restTiefe, Spielbrett brett,
                        Schiedsrichter richter) {
    int ermittelt = MAXIMUM;
    WertSpielzug besterZug = null;
    Spielzug[] moeglicheSpielzuege =
            richter.getAlleZuege(this.spiel.getSpielerB());
    for (Spielzug z : moeglicheSpielzuege) {
        Schiedsrichter kopie = richter.kopiere();
        PolySpielzug poly = new PolySpielzug();
        for (Spielfeld feld : z.getFelder()) {
            poly.erweitere(
                transformiere(feld, richter.getBrett(), kopie.getBrett())
            );
        }
        kopie.fuehreZugAus(poly, spiel.getSpielerB());
```

```
            int zugWert = 0;
            if (restTiefe <= 1
                || keinZugMoeglich(kopie, spiel.getSpielerB())) {
                zugWert = bewertung.analysiere(kopie.getBrett());
            } else {
              WertSpielzug zug =
                  maxWert(restTiefe - 1, kopie.getBrett(), kopie);
              zugWert = (zug != null) ? zug.getWert() : 0;
            }
            int zufall = (int) (Math.random() * 2);
            if (zugWert < ermittelt
                || (zugWert == ermittelt && zufall == 0)) {
                ermittelt = zugWert;
                besterZug = poly;
                besterZug.setWert(zugWert);
            }
        }
        return besterZug;
    }

    public boolean keinZugMoeglich(Schiedsrichter richter,
                                   Spieler spieler) {
        if (richter.getAlleZuege(spieler) != null
                && richter.getAlleZuege(spieler).length > 1) {
            return false;
        }
        return true;
    }

    private KoordinatenSpielfeld transformiere(Spielfeld feld,
                                               Spielbrett brett,
                                               Spielbrett kopie) {
        KoordinatenSpielfeld[] felder =
                new KoordinatenSpielfeld[kopie.getFelder().length];
        for (int i = 0; i < kopie.getFelder().length; i++) {
            felder[i] = (KoordinatenSpielfeld) kopie.getFelder()[i];
        }
        for (KoordinatenSpielfeld f : felder) {
            if (f.getReihe() == ((KoordinatenSpielfeld)feld).getReihe()
              && f.getSpalte() == ((KoordinatenSpielfeld)feld).getSpalte()) {
                return f;
            }
        }
        return null;
    }
}
```

4.4.6 Der Schiedsrichter

Bevor hier auf Implementierungen des Interface `Schiedsrichter` eingegangen werden können, müssen zunächst eine Hilfsklasse vorgestellt werden:
Die `FehlerZugException` wird geworfen, wenn ein Spielzug von einem ComputerSpieler eingegeben wird, der nicht den Spielregeln entspricht. Dann soll das Spiel sofort abgebrochen werden, da die Implementierung fehlerhaft ist.

```
package framework;

public class FehlerZugException extends RuntimeException { }
```

Die Klasse `AbstrakterSchiedsrichter` ist die Basisimplementierung für das `Schiedsrichter`-Interface.

```
package framework;

abstract public class AbstrakterSchiedsrichter implements Schiedsrichter {

    protected Spiel spiel;
    protected Interpreter interpreter;
    protected Spielbrett brett;

    public AbstrakterSchiedsrichter () {
        this.interpreter =
            new AufgabeDekorierer (new SpielzugInterpreter(this));
    }

    public Spielbrett getBrett() {
        return this.brett;
    }

    public void setBrett(Spielbrett brett) {
        this.brett = brett;
    }

    public Spiel getSpiel() {
        return spiel;
    }

    public void setSpiel (Spiel spiel) {
        this.spiel = spiel;
    }

    public Interpreter getInterpreter() {
        return interpreter;
    }

    public Spieler ermittleSpieler(Spieler vorherigerSpieler) {
```

```
        Spieler spielerA = this.spiel.getSpielerA();
        Spieler spielerB = this.spiel.getSpielerB();
        if (vorherigerSpieler == null
            || vorherigerSpieler.equals(spielerB)) {
            return spielerA;
        } else {
            return spielerB;
        }
    }

    public boolean zugKorrekt(Spielzug zug, Spieler spieler) {
        MessageDialog message = this.spiel.getMessageDialog();
        if (zug == null) {
            message.gebeAus("Es wurde kein gültiger Zug eingegeben.");
            return false;
        }
        Spielzug[] zuege = getAlleZuege(spieler);
        for (Spielzug z : zuege) {
            if (zug.equals(z)) {
                return true;
            }
        }
        if (!(spieler instanceof MenschSpieler)) {
            throw new FehlerZugException();
        }
        message.gebeAus("Der eingegebene Zug entspricht "
                                    + "nicht den Spielregeln.");
        return false;
    }
}
```

4.4.7 Das Spiel

Die Klasse `DefaultSpiel` ähnelt sehr der Klasse `SpielImpl` des Prototyps. Ein wesentlicher Unterschied ist die Erweiterung um eine sogenannte *History*. Alle Züge und Spielbrettsituationen werden gespeichert und können über die Methoden `getZugHistory` bzw. `getBrettHistory` abgerufen werden.

```
package framework;

import java.util.List;
import java.util.ArrayList;

public class DefaultSpiel implements Spiel {

    private Spieler spielerA;
    private Spieler spielerB;
    private Schiedsrichter richter;
    private SpielbrettView view;
```

```
private InputDialog inputDialog;
private MessageDialog messageDialog;
private List<Spielzug> zuege;
private List<Spielbrett> bretter;

public DefaultSpiel(Spieler spielerA, Spieler spielerB,
                    Schiedsrichter richter, SpielbrettView view,
                    InputDialog inputDialog,
                    MessageDialog messageDialog) {

    this.zuege = new ArrayList<Spielzug>();
    this.bretter = new ArrayList<Spielbrett>();
    this.spielerA = spielerA;
    this.spielerA.setSpiel(this);
    this.spielerB = spielerB;
    this.spielerB.setSpiel(this);
    this.richter = richter;
    this.richter.setSpiel(this);
    this.inputDialog = inputDialog;
    this.messageDialog = messageDialog;

    this.view = view;
    this.richter.getBrett().meldeAn(view);
    this.richter.getBrett().initialisiere();
}

public List<Spielzug> getZugHistory() {
    return this.zuege;
}

public List<Spielbrett> getBrettHistory() {
    return this.bretter;
}

public Spieler getSpielerA() {
    return this.spielerA;
}

public Spieler getSpielerB() {
    return this.spielerB;
}

public Schiedsrichter getRichter() {
    return this.richter;
}

public InputDialog getInputDialog() {
    return this.inputDialog;
```

```
    }

    public MessageDialog getMessageDialog() {
        return this.messageDialog;
    }

    public void steuereAblauf() {
        Spieler aktuellerSpieler = null;
        Figur figur;
        Spielzug zug;
        boolean zugKorrekt;
        do {
            aktuellerSpieler =
                    this.richter.ermittleSpieler(aktuellerSpieler);
            zug = aktuellerSpieler.getZug();
            this.zuege.add(zug);
            this.bretter.add(richter.getBrett());
            if (!this.richter.zugKorrekt(zug, aktuellerSpieler)) {
                break;
            }
            this.richter.fuehreZugAus(zug, aktuellerSpieler);
        } while (!(this.richter.spielBeendet()));
        this.richter.verkuendeSieger();
    }
}
```

4.4.8 Die Fabriken

In einer `main`-Methode müssen nun alle nötigen Objekte für ein Spiel erzeugt und das Spiel gestartet werden. Zur Objekterzeugung setzen wir das Entwurfsmuster Abstrakte Fabrik (siehe Kapitel 3.4, Seite 51) ein.
Für den Schiedsrichter und den View auf das Spielbrett stellen wir oberste Schnittstellen bereit, die die konkreten Fabriken implementieren müssen. Da das Framework unabhängig vom konkreten Spiel bzw. unabhängig vom konkreten View sein soll, macht es keinen Sinn, an diesen Stellen ein Default-Verhalten zu implementieren.
Zwar könnte der Hamstersimulator Default-View sein – das Problem wäre dann jedoch eine Schnittstellenimplementierung, die `SpielbrettView` zumindest so implementiert, dass möglichst viele Spiele abgebildet werden können. Da dies in dieser Arbeit nicht geleistet werden kann, muss auf einen Default-View verzichtet werden.

```
package framework;

public interface SchiedsrichterFabrik {
    public Schiedsrichter erzeugeSchiedsrichter();
}

package framework;

public interface SpielbrettViewFabrik {
```

```
    public SpielbrettView erzeugeSpielbrettView(Schiedsrichter richter);
}
```

Bevor wir uns als nächstes um die Erzeugung der Spielerobjekte kümmern können, müssen wir zunächst ein damit zusammenhängendes Problem lösen: Verschiedene Spielerklassen benötigen verschiedene Parameter für ihre Konstruktoren. Zum Beispiel ist für die Erzeugung eines `MinimaxSpieler`-Objektes ein `int`-Wert für die Suchtiefe und eine `Bewertung` nötig, diese Werde müssen jedoch bei der Erzeugung eines `MenschSpieler`-Objektes nicht angegeben werden.
Daraus folgt, dass in einer abstrakten Fabrik `SpielerFabrik` keine Methode `erzeugeSpieler` mit einer fest codierten Parameterliste definiert werden kann.
Wir lösen dieses Problem, in dem wir eine Klasse `FabrikParameter` verwenden, in der Parameter beliebiger Anzahl und beliebigen Typs gespeichert werden können. Wenn der Methode `erzeugeSpieler` dann eine Instanz dieser Klasse als Parameter mitgegeben wird, ist dies zumindest flexibel genug für die im Framework vordefinierten Spielerklassen.
Die Klasse `FabrikParameter` wird als Wrapper für eine `Map` eingesetzt, die Parameternamen auf konkrete Objekte abbildet. Außerdem werden `get`-Methoden für wahrscheinlich häufig vorkommende Parameter-Typen bereitgestellt:

```
package framework;

import java.util.Map;
import java.util.HashMap;

public class FabrikParameter {

    private Map<String, Object> parameters = new HashMap<String, Object>();

    public void setParam(String name, Object wert) {
        this.parameters.put(name, wert);
    }

    public Object getParam(String name) {
        return this.parameters.get(name);
    }

    public boolean getBoolParam(String name) {
        return (Boolean) this.parameters.get(name);
    }

    public int getIntParam(String name) {
        return (Integer) this.parameters.get(name);
    }

    public Bewertung getBewertungParam(String name) {
        return (Bewertung) this.parameters.get(name);
    }
}
```

Die oberste Schnittstelle für die Erzeugung der `Spieler`-Objekte schreibt die Implementierung einer Methode `erzeugeSpieler` vor, die – wie schon diskutiert – eine Instanz der Klasse `FabrikParameter` als Parameter übergeben bekommt. Da für *jeden* Spieler entschieden werden muss, ob er Spieler A oder Spieler B ist, wird dieser Parameter von der gekapselten `Map` getrennt aufgeführt.

```
package framework;

public interface SpielerFabrik {
    public Spieler erzeugeSpieler(boolean istSpielerA,
                                  FabrikParameter parameter);
}
```

Für die Erzeugung eines `MenschSpieler`-Objektes reicht der Parameter `istSpielerA` aus. Aus diesem Grund wird in der Methode `erzeugeSpieler` der `MenschSpielerFabrik` der angegebene `FabrikParameter` ignoriert.

```
package framework;

public class MenschSpielerFabrik implements SpielerFabrik {

    public Spieler erzeugeSpieler(boolean istSpielerA,
                                  FabrikParameter parameter) {
        return new MenschSpieler(istSpielerA);
    }
}
```

Die `erzeugeSpieler`-Methode der konkreten `MinimaxSpielerFabrik` überprüft die erhaltene Objektinstanz der Klasse `FabrikParameter`. Sind die dort enthaltenen Werte korrekt, wird ein `MenschSpieler`-Objekt zurückgegeben. Ist dies nicht so, wird eine `IllegalArgumentException` geworfen.

```
package framework;

public class MinimaxSpielerFabrik implements SpielerFabrik {

    public Spieler erzeugeSpieler(boolean istSpielerA,
                                  FabrikParameter parameter) {
        if (parameter == null) {
            return null;
        }
        Integer tiefe = parameter.getIntParam("tiefe");
        Bewertung bewertung = parameter.getBewertungParam("bewertung");
        if (tiefe == null || bewertung == null) {
            throw new IllegalArgumentException();
        }
        return new MinimaxSpieler(istSpielerA, tiefe, bewertung);
    }
}
```

Als oberste Schnittstelle für die Fabriken der Spielerzeugung wird das Interface `SpielFabrik` definiert:

```
package framework;

public interface SpielFabrik {

    public Spiel erzeugeSpiel(MessageDialog messageDialog,
                              InputDialog inputDialog,
                              SchiedsrichterFabrik schiedsrichterFabrik,
                              SpielbrettViewFabrik spielbrettViewFabrik,
                              SpielerFabrik spielerFabrikA,
                              FabrikParameter parameterA,
                              SpielerFabrik spielerFabrikB,
                              FabrikParameter parameterB);
}
```

```
package framework;

public class DefaultSpielFabrik implements SpielFabrik {

    public Spiel erzeugeSpiel(MessageDialog messageDialog,
                              InputDialog inputDialog,
                              SchiedsrichterFabrik schiedsrichterFabrik,
                              SpielbrettViewFabrik spielbrettViewFabrik,
                              SpielerFabrik spielerFabrikA,
                              FabrikParameter parameterA,
                              SpielerFabrik spielerFabrikB,
                              FabrikParameter parameterB) {

        Schiedsrichter richter =
                schiedsrichterFabrik.erzeugeSchiedsrichter();
        SpielbrettView brettView =
                spielbrettViewFabrik.erzeugeSpielbrettView(richter);
        Spieler spielerA = spielerFabrikA.erzeugeSpieler(true, parameterA);
        Spieler spielerB = spielerFabrikB.erzeugeSpieler(false, parameterB);
        return new DefaultSpiel(spielerA, spielerB, richter, brettView,
                                inputDialog, messageDialog);
    }
}
```

4.5 Ausblick

In diesem Kapitel soll der Frage nachgegangen werden, wie dieses Framework weiter ausgebaut und verbessert werden kann.

- Bei der Implementierung von Spielen mit Hilfe des Frameworks muss zunächst einmal geprüft werden, ob das Framework überhaupt für das Spiel ausgelegt ist. Ist dies nicht der Fall, muss das Framework gegebenenfalls angepasst werden.

- Für alle implementierten Spiele sollte es zumindest einen View geben, der sie darstellen kann. Bis jetzt haben wir nur einen View kennengelernt, der mit Hilfe des Hamstersimulators TicTacToe (und *nur* TicTacToe) darstellen kann. In Kapitel 5 werden wir kennenlernen, wie man den Hamstersimulator verwenden kann, um das Damespiel darzustellen.

- Es gibt noch eine Fülle an weiteren Spielalgorithmen neben dem Minimax-Algorithmus bzw. Verfeinerungen, um diesen zu verbessern. Diese könnten nach und nach implementiert werden.

- In der jetzigen Version sind wir auf genau zwei Spieler festgelegt. Das Framework könnte so erweitert werden, dass eine beliebige Anzahl von Spielern möglich ist. Die wesentlichste Änderung wäre dann, die Spieleridentidfikation nicht mehr über ein Attribut vom Typ `boolean` laufen zu lassen, sondern über eine Identifikationsnummer, z.B. vom Typ `int`.

- Um so mehr Spiele implementiert werden, um so mehr Abstraktionen sind auch möglich. Ein Beispiel: Die Spiele TicTacToe, Vier Gewinnt und Fünf Gewinnt haben alle gemeinsam, dass man eine bestimmte Anzahl von Figuren in eine Reihe, Spalte oder Diagonale bringen muss. Für diese Spiele könnte man eine parametrisierte Prüfmethode im Framework implementieren, die diese nur noch zu benutzen brauchen.

Eine weitere – aber ganz andere – Möglichkeit, das Framework zu erweitern, soll an dieser Stelle nicht außer acht gelassen werden: Um so komplexer das Framework wird, um so anfälliger wird es für Fehler. Deswegen ist es unabkömmlich, Sinn und Zweck aller Klassen und Methoden genau zu spezifizieren, zu dokumentieren und zu testen.

5 Implementierung des DameSpiels

5.1 Die Dame-Bewertung

Für das DameSpiel wurde die `DefaultBewertung` so erweitert, dass für eine Dame eines Spielers ein Punkt mehr dazu gezählt wird.
Im Konstruktor findet folgender Mechanismus statt: Die Bewertung kontrolliert, ob sie das angegebene Spiel überhaupt sinnvoll bewerten kann und wirft im negativen Fall eine `IllegalArgumentExceptio`
Der Programmierer hat in einer `String`-Konstanten anzugeben, welche Spiele von der Bewertung bewertet werden können.

```
package dame;

import java.util.ArrayList;
import framework.Spielbrett;
import framework.Spielfeld;
import framework.Schiedsrichter;
import framework.Figur;
import framework.DefaultBewertung;

public class DameBewertung extends DefaultBewertung {

    public static final String[] moeglicheSpiele = { "Dame" };

    public DameBewertung (Schiedsrichter richter) {
        boolean bewertbar = false;
        for (String str : moeglicheSpiele) {
            if (richter.toString().equals(str)) {
                bewertbar = true;
            }
        }
        if (!bewertbar) {
            throw new IllegalArgumentException();
        }
    }

    public int analysiere(Spielbrett brett) {
        int ergebnis = 0;
        ergebnis = super.analysiere(brett);
        Spielfeld[] felder = brett.getFelder();
        //Anzahl der Damen
        for (Spielfeld feld : felder) {
            Figur figur = feld.getFigur();
            if (figur != null && figur instanceof DameDame) {
                if (figur.getSpieler().istSpielerA()) {
                    ergebnis++;
                } else {
                    ergebnis--;
                }
```

```
            }
        }
        return ergebnis;
    }
}
```

5.2 Die Damefigur

Es gibt zwei Sorten von Figuren im Damespiel: Den normalen Damestein und die Dame. Mangels besserer Namen wurden für ihre Klassen `DameStein` und `DameDame` genannt. Sie erben von einer gemeinsam Oberklasse `DameFigur`, die einige gemeinsame Funktionalitäten implementiert.

```
package dame;

import java.util.List;
import java.util.ArrayList;

import framework.BasisKoordinatenFigur;
import framework.Spieler;
import framework.Spielzug;
import framework.ZweierSpielzug;
import framework.PolySpielzug;
import framework.Spielbrett;
import framework.Figur;
import framework.Schiedsrichter;
import framework.KoordinatenSpielbrett;
import framework.KoordinatenSpielfeld;
import framework.Spielfeld;

abstract public class DameFigur extends BasisKoordinatenFigur {

    public DameFigur(Spieler spieler, Spielfeld feld) {
        super(spieler, (KoordinatenSpielfeld) feld);
    }

    public List<Spielzug> moeglicheSprungZuege(PolySpielzug zug,
                                               List<Spielzug> list) {
        Spielzug[] aktuelleZuege = moeglicheSpruenge();
        if (aktuelleZuege != null && aktuelleZuege.length > 0) {
            for (Spielzug z : aktuelleZuege) {
                Schiedsrichter kopie =
                    this.feld.getBrett().getRichter().kopiere();
                KoordinatenSpielfeld[] felder =
                    (KoordinatenSpielfeld[]) kopie.getBrett().getFelder();
                zug.erweitere(z.getZiel());
                z = new ZweierSpielzug(
                      transformiere(
                        z.getStart(), this.feld.getBrett(), kopie.getBrett()
```

```
                ),
                transformiere(
                  z.getZiel(),this.feld.getBrett(),kopie.getBrett()
                )
            );
            kopie.fuehreZugAus(z, getSpieler());
            DameFigur kopieFigur = null;
            KoordinatenSpielfeld f = (KoordinatenSpielfeld) z.getZiel();
            kopieFigur = (DameFigur) f.getFigur();
            if (kopieFigur != null) {
                list = kopieFigur.moeglicheSprungZuege (zug, list);
            }
        }
    } else {
        if (zug.getFelder().length > 1) {
            list.add (zug);
        }
    }
    return list;
}

protected Spielzug moeglicherSprung(int reihe, int spalte) {
    DameSpielbrett brett = (DameSpielbrett) this.feld.getBrett();
    int reihe2 = (reihe > 0) ? 2 : -2;
    int spalte2 = (spalte > 0) ? 2 : -2;
    if (feldFeindFigur(getX() + reihe, getY() + spalte)) {
        if (feldFrei(getX() + reihe2, getY() + spalte2)) {
            KoordinatenSpielbrett xybrett =
                (KoordinatenSpielbrett) brett;
            return new ZweierSpielzug (this.feld,
                                       xybrett.sucheFeld (getX()
                                         + reihe2,
                                       getY() + spalte2));
        }
    }
    return null;
}

abstract protected Spielzug[] moeglicheSpruenge();

abstract public List<Spielzug> moeglicheSpielzuege();

abstract public DameFigur kopiere();

private KoordinatenSpielfeld transformiere(Spielfeld feld,
                                           Spielbrett brett,
                                           Spielbrett kopie) {
    KoordinatenSpielfeld[] felder =
```

```
            (KoordinatenSpielfeld[]) kopie.getFelder();
        for (KoordinatenSpielfeld f : felder) {
            if (f.getReihe() == ((KoordinatenSpielfeld)feld).getReihe()
              && f.getSpalte() == ((KoordinatenSpielfeld)feld).getSpalte()) {
                return f;
            }
        }
        return null;
    }
}
```

5.3 Der Damestein

```
package dame;

import java.util.List;
import java.util.ArrayList;

import framework.Spieler;
import framework.KoordinatenSpielbrett;
import framework.Spielzug;
import framework.Spielfeld;
import framework.ZweierSpielzug;
import framework.Figur;

public class DameStein extends DameFigur{

    public DameStein(Spieler spieler, Spielfeld feld) {
        super (spieler, feld);
    }

    public DameFigur kopiere() {
        return new DameStein (this.spieler, this.feld);
    }

    public String toString() {
        return "Damestein";
    }

    public List<Spielzug> moeglicheSpielzuege() {
        DameSpielbrett brett = (DameSpielbrett) this.feld.getBrett();
        List<Spielzug> list = new ArrayList<Spielzug>();
        Spielfeld[] felder = brett.getFelder();
        int reihe = istA() ? -1 : 1;
        if (feldFrei(getX() + reihe, getY() + 1)) {
            list.add(new ZweierSpielzug(this.feld,
                                        brett.sucheFeld(getX() + reihe,
                                        getY() + 1)));
```

```
        }
        if (feldFrei(getX() + reihe, getY() - 1)) {
            list.add(new ZweierSpielzug(this.feld,
                                        brett.sucheFeld(getX() + reihe,
                                        getY() - 1)));
        }
        return list;
    }

    protected Spielzug[] moeglicheSpruenge () {
        DameSpielbrett brett = (DameSpielbrett) this.feld.getBrett();
        List<Spielzug> list = new ArrayList<Spielzug>();
        int reihe = istA() ? -1 : 1;
        Spielzug sprungRechts = moeglicherSprung(reihe, 1);
        if (sprungRechts != null) {
            list.add(sprungRechts);
        }
        Spielzug sprungLinks = moeglicherSprung (reihe, -1);
        if (sprungLinks != null) {
            list.add(sprungLinks);
        }
        return list.toArray(new Spielzug[list.size()]);
    }
}
```

5.4 Die Dame

```
package dame;

import java.util.List;
import java.util.ArrayList;

import framework.Spieler;
import framework.Spielbrett;
import framework.Spielzug;
import framework.Spielfeld;
import framework.ZweierSpielzug;
import framework.BasisKonstanten;
import framework.Figur;

public class DameDame extends DameFigur implements BasisKonstanten {

    public DameDame(Spieler spieler, Spielfeld feld) {
        super(spieler, feld);
    }

    public DameFigur kopiere() {
        return new DameDame(this.spieler, this.feld);
```

```
    }

    public String toString() {
        return "DameDame";
    }

    public List<Spielzug> moeglicheSpielzuege() {
        DameSpielbrett brett = (DameSpielbrett) this.feld.getBrett();
        List<Spielzug> list = new ArrayList<Spielzug>();
        Spielfeld[] felder = brett.getFelder();
        for (int[] d : DIAGONAL_1) {
            if (feldFrei (getX() + d[0], getY() + d[1])) {
                list.add (new ZweierSpielzug(this.feld,
                                             brett.sucheFeld(getX() + d[0],
                                             getY() + d[1])));
            }
        }
        return list;
    }

    protected Spielzug[] moeglicheSpruenge () {
        DameSpielbrett brett = (DameSpielbrett) this.feld.getBrett();
        List<Spielzug> list = new ArrayList<Spielzug>();
        for (int[] d : DIAGONAL_1) {
            Spielzug sprung = moeglicherSprung(d[0], d[1]);
            if (sprung != null) {
                list.add(sprung);
            }
        }
        return list.toArray(new Spielzug[list.size()]);
    }
}
```

5.5 Das Dame-Spielbrett

Das `DameSpielbrett` erbt vom `KoordinatenSpielbrett` und implementiert zusätzlich die Startaufstellung der Figuren:

```
package dame;

import java.util.List;
import java.util.ArrayList;

import framework.Spielbrett;
import framework.KoordinatenSpielbrett;
import framework.Spieler;
import framework.Spielfeld;
import framework.Figur;
import framework.Schiedsrichter;
```

```
import framework.KoordinatenSpielfeld;

public class DameSpielbrett extends KoordinatenSpielbrett {

    public static final int[][] FIGUR_POSITIONEN_SPIELER_A
            = { {7, 0}, {7, 2}, {7, 4}, {7, 6},
                {6, 1}, {6, 3}, {6, 5}, {6, 7},
                {5, 0}, {5, 2}, {5, 4}, {5, 6} };

    public static final int[][] FIGUR_POSITIONEN_SPIELER_B
            = { {0, 1}, {0, 3}, {0, 5}, {0, 7},
                {1, 0}, {1, 2}, {1, 4}, {1, 6},
                {2, 1}, {2, 3}, {2, 5}, {2, 7} };

    public DameSpielbrett(DameSchiedsrichter richter) {
        super(richter, 8, 8);
    }

    public void initialisiere() {
        Spieler spielerA = this.richter.getSpiel().getSpielerA();
        Spieler spielerB = this.richter.getSpiel().getSpielerB();
        for (int[] koordinaten : FIGUR_POSITIONEN_SPIELER_A) {
            int reihe = koordinaten[0];
            int spalte = koordinaten[1];
            Spielfeld feld = sucheFeld(reihe, spalte);
            weiseZu(feld, new DameStein (spielerA, feld));
        }
        for (int[] koordinaten : FIGUR_POSITIONEN_SPIELER_B) {
            int reihe = koordinaten[0];
            int spalte = koordinaten[1];
            Spielfeld feld = sucheFeld(reihe, spalte);
            weiseZu(feld, new DameStein (spielerB, feld));
        }
        for (Spielfeld feld : felder) {
            if (feld.getFigur() != null) {
                feld.getFigur().setFeld(feld);
            }
        }
        benachrichtige();
    }

    public Spielbrett kopiere(Schiedsrichter richter) {
        Spielbrett kopie = new DameSpielbrett((DameSchiedsrichter) richter);
        kopie.setFelder(new KoordinatenSpielfeld[64]);
        Spielfeld[] kopieFelder = kopie.getFelder();
        int counter = 0;
        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
```

```
                kopieFelder[counter] =
                    new KoordinatenSpielfeld(kopie, i, j);
                counter++;
            }
        }
        for (int i = 0; i < this.felder.length; i++) {
            Spielfeld kopieFeld = kopieFelder[i];
            Figur figur = this.felder[i].getFigur();
            if (figur != null) {
                Figur k = ((DameFigur)figur).kopiere();
                kopieFeld.setFigur(k);
                k.setFeld (kopieFeld);
            }
        }
        return kopie;
    }
}
```

5.6 Der Dame-Schiedsrichter

```
package dame;

import java.util.ArrayList;
import java.util.List;
import java.util.LinkedList;
import java.util.StringTokenizer;

import framework.Schiedsrichter;
import framework.AbstrakterSchiedsrichter;
import framework.Spieler;
import framework.Spielbrett;
import framework.Spielfeld;
import framework.Spielzug;
import framework.ZweierSpielzug;
import framework.PolySpielzug;
import framework.Figur;
import framework.BasisKonstanten;
import framework.DefaultSpielbrett;
import framework.MessageDialog;
import framework.KoordinatenSpielfeld;
import framework.Spiel;

public class DameSchiedsrichter extends AbstrakterSchiedsrichter
                               implements BasisKonstanten {

    private DameSpielbrett brett;
    private Spieler sieger;
```

```
public DameSchiedsrichter() {
    super();
    this.brett = new DameSpielbrett(this);
}

public DameSchiedsrichter(Spielbrett brett) {
    super();
    this.brett = (DameSpielbrett) brett;
}

public Schiedsrichter kopiere() {
    DameSchiedsrichter kopie = new DameSchiedsrichter(this.brett);
    kopie.setSpiel(this.getSpiel());
    kopie.setBrett((DameSpielbrett) brett.kopiere(kopie));
    return kopie;
}

public String toString() {
    return "Dame";
}

public Spielzug[] alleSprungzuege(Spieler spieler) {
    List<Spielzug> list = new ArrayList<Spielzug>();
    Spielfeld[] felder = this.brett.getFelder();
    for (Spielfeld feld : felder) {
        DameFigur f = (DameFigur) feld.getFigur();
        if (f == null) {
            continue;
        }
        if (!(f.getSpieler().equals(spieler))) {
            continue;
        }
        list.addAll(f.moeglicheSprungZuege(new PolySpielzug (feld),
                                           new LinkedList<Spielzug>())
                    );
    }
    if (list != null && list.size() > 0) {
        return list.toArray(new Spielzug[list.size()]);
    } else {
        return null;
    }
}

public Spielzug[] getAlleZuege(Spieler spieler) {
    Spielzug[] sprungzuege = alleSprungzuege(spieler);
    if (sprungzuege != null && sprungzuege.length > 0) {
        return sprungzuege;
    }
```

```
    List<Spielzug> list = new ArrayList<Spielzug>();
    Spielfeld[] felder = this.spiel.getRichter().getBrett().getFelder();
    this.brett = (DameSpielbrett) this.spiel.getRichter().getBrett();
    for (Spielfeld feld : felder) {
        Figur f = feld.getFigur();
        if (f == null) {
            continue;
        }
        if (!(f.getSpieler().equals(spieler))) {
            continue;
        }
        list.addAll(((DameFigur)f).moeglicheSpielzuege());
    }
    return list.toArray(new Spielzug[list.size()]);
}

public void fuehreZugAus(Spielzug zug, Spieler spieler) {
    if (zug.getFelder().length == 2) {
        fuehreZweierZugAus(zug, spieler);
    } else {
        for (int i = 0; i < (zug.getFelder().length - 1); i++) {
            fuehreZweierZugAus(new ZweierSpielzug(zug.getFelder()[i],
                                                  zug.getFelder()[i+1]),
                                                  spieler);
        }
    }
}

public void fuehreZweierZugAus(Spielzug zug, Spieler spieler) {
    KoordinatenSpielfeld sk = (KoordinatenSpielfeld) zug.getStart();
    KoordinatenSpielfeld zk = (KoordinatenSpielfeld) zug.getZiel();
    DameFigur f = (DameFigur) sk.getFigur();
    this.brett = (DameSpielbrett) this.spiel.getRichter().getBrett();
    this.brett.weiseZu(zk, f);
    for (int i = 0; i < DIAGONAL_1.length; i++) {
        if ((sk.getReihe() + DIAGONAL_2[i][0] == zk.getReihe())
             && (sk.getSpalte() + DIAGONAL_2[i][1] == zk.getSpalte())) {
            Spielfeld loeschfeld =
                brett.sucheFeld((sk.getReihe() + DIAGONAL_1[i][0]),
                                (sk.getSpalte() + DIAGONAL_1[i][1]));
            if (loeschfeld.getFigur() != null) {
                this.brett.weiseZu(loeschfeld, null);
                break;
            }
        }
    }
    if (zug.getZiel().getFigur() != null) {
        if (zug.getZiel().getFigur() instanceof DameStein) {
```

```
            if (spieler.istSpielerA()
              && ((KoordinatenSpielfeld)zug.getZiel()).getReihe() == 0) {
                this.brett.wandleUm(zug.getZiel().getFigur(),
                                new DameDame (this.spiel.getSpielerA(),
                                              zug.getZiel()));
            } else if (!(spieler.istSpielerA())
                     && ((KoordinatenSpielfeld)zug.getZiel()).getReihe()
                           == 7) {
                this.brett.wandleUm(zug.getZiel().getFigur(),
                                    new DameDame(this.spiel.getSpielerB(),
                                                 zug.getZiel())
                                     );
            }
        }
    }
}

public boolean spielBeendet() {
    Spielfeld[] felder = this.spiel.getRichter().getBrett().getFelder();
    boolean spielerA = false;
    boolean spielerB = false;
    for (Spielfeld feld : felder) {
        Figur figur = feld.getFigur();
        if (figur != null) {
            if (figur.getSpieler().istSpielerA()) {
                spielerA = true;
                this.sieger = this.spiel.getSpielerA();
            } else {
                spielerB = true;
                this.sieger = this.spiel.getSpielerA();
            }
        }
        if (spielerA & spielerB) {
            return false;
        }
    }
    return true;
}

public void verkuendeSieger () {
    MessageDialog message = this.spiel.getMessageDialog();
    if (sieger == null) {
        message.gebeAus("Spielabbruch. Es steht unentschieden.");
    } else {
        if (this.sieger.istSpielerA()) {
            message.gebeAus("Spieler A ist Sieger!");
        } else {
            message.gebeAus("Spieler B ist Sieger!");
```

```
            }
        }
    }

    private boolean istSprungZug(Spielzug zug, Spielzug[] sprungzuege) {
        if (zug == null) {
            return false;
        }
        if (sprungzuege == null) {
            return false;
        }
        for (int i = 0; i < sprungzuege.length; i++) {
            if (zug.equals(sprungzuege[i])) {
                return true;
            }
        }
        return false;
    }
}
```

5.7 Der Hamster-Dame-View

Um dieses Spiel mit dem Hamstersimulator darstellen zu können, wird zuerst eine Hilfsklasse `HamsterFigur` implementiert:

```
package hamsterView;

public class HamsterFigur extends AllroundHamster {

    public HamsterFigur(int r, int s, int b, int k) {
        super(r, s, b, k);
    }

    public void bewege(int reihe, int spalte) {
        int b = getBlickrichtung();
        super.gotoKachel(reihe, spalte);
        setzeBlickrichtung(b);
    }
}
```

Die Klasse `HamsterDameView` erzeugt im Konstruktor fünf „Zuschauerhamster“, so dass die Hamster, die die Spielfiguren repräsentieren, durchgängig weiß sind. Es wird hier derselbe Mechanismus wie schon bei der Bewertung eingesetzt: Auch der View schaut nach, ob er das Spiel überhaupt sinnvoll darstellen kann. Wenn nicht, wirft er eine `RuntimeException`.

```
package hamsterView;

import java.util.Map;
import java.util.HashMap;
```

```
import framework.Spielbrett;
import framework.Spielfeld;
import framework.Figur;
import framework.KoordinatenSpielfeld;
import framework.SpielbrettView;
import framework.Schiedsrichter;

import dame.DameStein;
import dame.DameDame;

public class HamsterDameView implements SpielbrettView {

    private String[] bilder;
    private Map<Figur, HamsterFigur> hamsterFiguren;
    public static final String[] moeglicheSpiele = { "Dame" };

    public HamsterDameView(Schiedsrichter richter) {
        boolean darstellbar = false;
        for (String str : moeglicheSpiele) {
             if (richter.toString().equals(str)) {
                 darstellbar = true;
             }
        }
        if (!darstellbar) {
            throw new RuntimeException();
        }
        this.hamsterFiguren =
            new HashMap<Figur, HamsterFigur>();
        //Zuschauer:
        Hamster julien = new Hamster(0, 1, Hamster.SUED, 0);
        Hamster dick = new Hamster(1, 0, Hamster.OST, 0);
        Hamster george = new Hamster(0, 2, Hamster.SUED, 0);
        Hamster ann = new Hamster(2, 0, Hamster.OST, 0);
        Hamster timmyDerHun = new Hamster(0, 3, Hamster.SUED, 0);
    }

    public void aktualisiere(Spielbrett brett) {
        Spielfeld[] felder = brett.getFelder();
        boolean neuerHamster = false;
        for (int i = 0; i < felder.length; i++) {
            Figur figur = felder[i].getFigur();
            if (figur != null) {
                int x = ((KoordinatenSpielfeld) felder[i]).getReihe();
                int y = ((KoordinatenSpielfeld) felder[i]).getSpalte();
                if (!hamsterFiguren.containsKey(figur)) {
                    Hamster[] alleHamster =
                        Territorium.getHamster(x + 2, y + 2);
```

```
                    HamsterFigur hamster = null;
                    if (alleHamster == null || alleHamster.length < 1) {
                        int blick = (figur.getSpieler().istSpielerA())
                                    ? Hamster.NORD
                                    : Hamster.SUED;
                        hamster = new HamsterFigur(x + 2, y + 2, blick, 0);
                        hamsterFiguren.put(figur, hamster);
                        neuerHamster = true;
                    } else {
                        hamster = (HamsterFigur) alleHamster[0];
                        hamster.linksUm();
                        hamsterFiguren.put(figur, hamster);
                    }
                } else {
                    HamsterFigur hamster = hamsterFiguren.get(figur);
                    if (x != hamster.getReihe() - 2
                        || y != hamster.getSpalte() - 2) {
                        hamster.bewege (x + 2, y + 2);
                    }
                }
            }
        }
        if (!neuerHamster) {
            for (int i = 0; i < felder.length; i++) {
                Figur figur = felder[i].getFigur();
                if (figur == null) {
                    int x =
                        ((KoordinatenSpielfeld) felder[i]).getReihe();
                    int y =
                        ((KoordinatenSpielfeld) felder[i]).getSpalte();
                    Hamster[] alleHamster =
                        Territorium.getHamster(x + 2, y + 2);
                    HamsterFigur hamster = null;
                    if (alleHamster != null && alleHamster.length > 0) {
                        hamster = (HamsterFigur) alleHamster[0];
                    }
                    if (hamster != null) {
                        hamster.bewege(11, 11);
                    }
                }
            }
        }
    }
}
```

5.8 Das Hauptprogramm

Im Hauptprogramm werden sämtliche benötigen Objekte erzeugt – teilweise durch Fabriken –, und dann der Spielablauf gestartet.

```
import framework.*;
import hamsterView.*;
import dame.*;

void main() {
    MessageDialog messageDialog = new HamsterMessageDialog();
    InputDialog inputDialog = new HamsterInputDialog();
    SchiedsrichterFabrik richterFabrik = new DameSchiedsrichterFabrik();
    SpielbrettViewFabrik viewFabrik = new HamsterViewFabrik();
    SpielerFabrik spielerAFabrik = new MenschSpielerFabrik();
    SpielerFabrik spielerBFabrik = new MenschSpielerFabrik();
    //spielerA
    FabrikParameter parameterA = new FabrikParameter();

    //spielerB
    FabrikParameter parameterB = new FabrikParameter();

    SpielFabrik fabrik = new DefaultSpielFabrik();

    Spiel spiel = fabrik.erzeugeSpiel(messageDialog,
                                      inputDialog,
                                      richterFabrik,
                                      viewFabrik,
                                      spielerAFabrik, parameterA,
                                      spielerBFabrik, parameterB);
    spiel.steuereAblauf();

}
```

6 Fazit

Mein Hauptziel war es, einem Schüler oder Studenten, der gerade die Grundlagen der objektorientierten Programmierung erlernt hat, Programmier- und Entwurfskonzepte zu zeigen, an die er mit seinem Wissen direkt anknüpfen kann.
Dabei ging es nicht darum, vollständig alles zu erklären: Zum Beispiel wird in dieser Arbeit nur ein winziger Teil der UML verwendet und nur sehr oberflächlich beschrieben. Aber der Leser bekommt einen Eindruck davon, wie man sie praktisch einsetzen kann und das ist es, worauf es mir ankam. Auch die Java-Klassenbibliothek wird nicht umfassend behandelt, sondern es werden nur einige Aspekte vorgestellt und wie man diese für die eigenen Programme verwenden kann.

Das Herzstück dieser Arbeit ist das dritte Kapitel mit den Hamsterprogrammen zu den einzelnen Entwurfsmustern. Ich habe hier versucht, einen Kompromiss zu machen zwischen dem Fehler, alles ausschweifend erklären zu wollen und dabei das eigentliche Ziel aus den Augen zu verlieren und dem Fehler, alles so konzentriert und kompakt darzustellen, dass es der Anfänger nicht mehr versteht.
Nach dem Durcharbeiten eines jeden Kapitels sollte jeder Leser (mit den Vorkenntnissen, die ich voraussetze) von sich sagen können: „Ich habe verstanden, worum es bei dem Muster X geht!“ Sollte dies nicht der Fall sein, so habe ich mein Ziel verfehlt.

Der Schwierigkeitsgrad von Kapitel 4 und 5 liegt meiner Ansicht nach deutlich über dem von Kapitel 3. Dies liegt auch daran, weil ich vieles offen lasse: Die Anforderungsdefinition wurde nur soweit betrieben, so dass man einen Entwurf wagen konnte. Der Entwurf enthält kaum Diagramme, dafür viele Entwurfsüberlegungen und -entscheidungen und daraus resultierende Interfaces. Diese Thematik wird aber ebenfalls nicht umfassend und abschließend behandelt, sondern soll eine Ideensammlung sein. Es geht hier also nicht um die Vorstellung eines perfekten Entwurfs für ein Spiele-Framework, sondern vielmehr um das Vorstellen von Gedankengängen und Überlegungen, die bei dem Entwurf eines Spiele-Frameworks bedacht werden müssen oder zumindest eine Rolle spielen könnten.
Hier lag der Schwerpunkt also nicht auf dem konkreten Vermitteln von Faktenwissen wie bei Kapitel 3. Es ging hier zunächst darum, den Programmieranfänger mit einer so komplexen Aufgabe zu konfrontieren, so dass er sie mit seinen Mitteln nicht überblicken kann. Als nächstes wurde dann Schritt für Schritt gezeigt, wie man mit den bereits zur Verfügung stehenden Mitteln – Entwurfsmustern, UML, Klassenbibliothek – versuchen kann, mit diesem Problem umzugehen.

7 Literaturverzeichnis

[AIS+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King und Shlomo Angel: A Pattern Language. Oxford University Press, 1977.

[ASU99] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullmann: Compilerbau. Teil 1. 2.Auflage. Oldenbourg, 1999.

[Bal01] Helmut Balzert: Lehrbuch der Software-Technik. 2.Auflage. Spektrum-Verlag, 2001.

[Bol02] Dietrich Boles: Programmieren spielend gelernt mit dem Java-Hamster-Modell. 2.Auflage. Teubner, 2002.

[BB04] Dietrich Boles, Cornelia Boles: Objektorientierte Programmierung spielend gelernt mit dem Java-Hamster-Modell. Teubner, 2004.

[Bre05] Ulrich Breymann: C++. Einführung und professionelle Programmierung. 8. Auflage. Carl Hanser Verlag, 2005.

[BMMM98] William J. Brown, Raphael C. Malveau, Hays W. „Skip“ McCormick III, Thomas J. Mowbray: Anti Patterns. Refactoring Software, Architectures and Projects in Crisis. Wiley Computer Publishing, 1998.

[BMRSM00] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: Patternorientierte Softwarearchitektur. Ein Pattern-System. Addison-Wesley, 2000.

[Ess04] Friedrich Esser: Java 2. Patterns, Idioms, Java-Zertifizierung. Galileo Computing, 2001. (openBook online: www)

[GHJV04] Erich Gamma, Richard Helm, Ralph Johnson und Vlissides: Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software. Addison-Wesley, 2004.

[PBG04] Torsten Posch, Klaus Birken und Michael Gerdom: Basiswissen Softwarearchitektur. Verstehen, entwerfen, bewerten und dokumentieren. dpunkt-Verlag, 2004.

[Ull04] Christian Ullenboom: Java ist auch eine Insel. Programmieren für die Java 2-Plattform in der Version 5 (Tiger-Release). Galileo Computing, 2004.

[ZGK04] Wolfgang Zuser, Thomas Grechenig und Monika Köhle: Software Engineering mit UML und dem Unified Process. Pearson, 2004.

Versicherung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen verwendet habe.

Oldenburg, den 7. September 2005

Florian Marwede